Exercises - Authentication Strategies

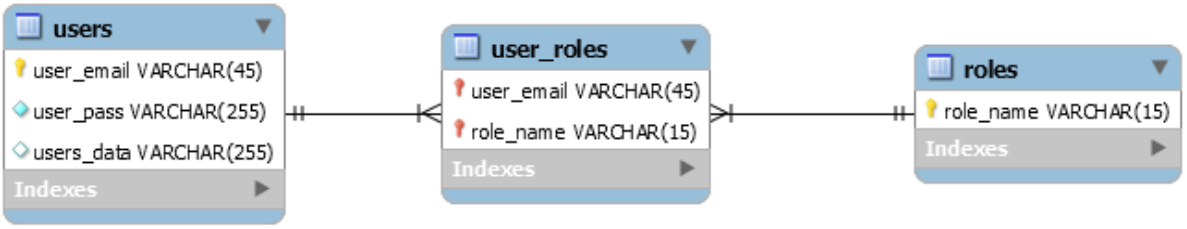
When these exercises are over you will know how to explain and demonstrate:

- Selecting the right Authentication strategy for problems like:
    - A Traditional WEB,
    - An API (REST),
    - Backend for a Mobile APP
    - Your app requires access to a third party api
    - A remote server talks to your API
- How to Prevent Brute Force Attacks
- Using an existing “mature” framework
- Explain the password policy you have implemented
- Explain how you expect to treat users passwords
- Explain how you expect to handle "lost passwords" scenarios
- Explain how you expect to Secure User Names Passwords and Session Cookies (or tokens) in transit
- When and how to re-authenticate for important actions

For everything that follows, you are free to pick another server technology (JavaScript, Python etc.), but you will be guided through a Java/Tomcat/Jersey solution.

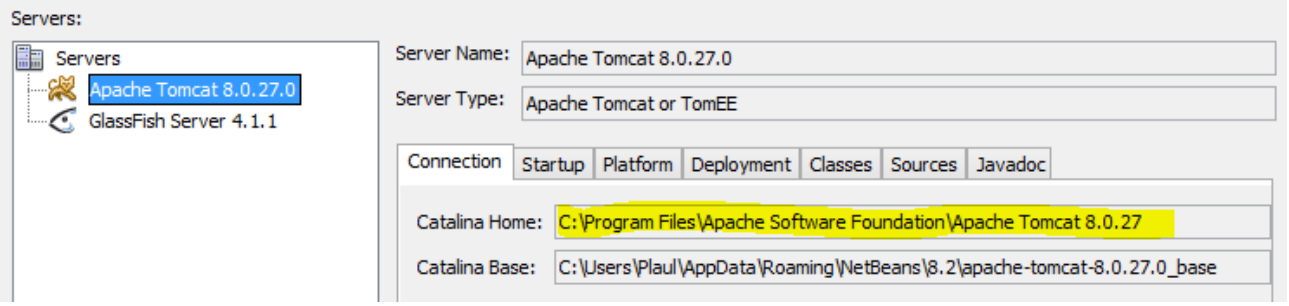
Getting ready for the exercises (will be done in the class)

All the following exercises uses this simple database design, inspired by this article.
It represents a simple role-based user setup, where users are stored with their email, a password + a simple text string to mimic their personal data.
Roles are kept in a separate table, with a many to many relationship between the two tables.



a) *Create a MySQL database and use this script to set up tables and a few test users:*
https://github.com/SecurityDatFall2018/Week-6-Authentication/blob/master/userDbScript.sql

b) *For the first exercises, Tomcat and not you, will be in charge of the authentication. So Tomcat needs access to the database driver (mysql-connector-java-xxxxxxx).*
In NetBeans Services tab, locate your Tomcat Server, and the path for Catalina Home (se below).



*In this folder, you will find a sub-folder **lib**.*
Copy the driver into this folder (it's still needed in the project for later exercises which use JPA.
IMPORTANT: When/if you deploy to Digital Ocean you need to install the driver here as well.

The purpose with the next two exercises is to demonstrate Basic/Form-based Authentication, and more important, how you can implement security with almost no effort (leave security to the server). That is, you rely on that someone (the inventors of the Java Web framework) can do security better that you can ;-)

Important: No matter what framework you are using, look for PROVEN and tried out solutions before inventing you own solution

Warning: Playing around with HTTP Basic Authentication often gets confusing because you cannot log out. So for all the following Basic exercises, open an incognito window (and ONLY one) while you test, and close it when you are done.

1) Basic Authentication (we will do this in the class)
Create a new Maven Web-project with NetBeans and do the following:
1a)
Setup Tomcat to use the database created above as a JDBC-realm (see here for details)
In "Web Pages" → META-INF locate the file context.xml and add the following content.
If you have used the script, provided above, you should only need to change the values marked in yellow (name of your db, user name and password for the db)

```
<Context path="/xxxxxxx">
    <Realm className="org.apache.catalina.realm.JDBCRealm"
    driverName="com.mysql.jdbc.Driver"
    connectionURL="jdbc:mysql://localhost:3306/dbName?user=dbUser&amp;password=dbPw"
    userTable="users"
    userNameCol="user_email"
    userCredCol="user_pass"
    userRoleTable="user_roles"
    roleNameCol="role_name"/>
</Context>
```

b)
Create two new static pages: `user.html` and `admin.html`, and link to the pages from `index.html`. Just add some dummy content so we now what the pages "represents".

c) Create a `web.xml` file in the project (just use the wizard, you should have done this before)

d) Now (declaratively) secure the pages so that only authenticated users with:

- **The user role**: can access the user.html page
- **The admin role**: can access the admin.html page
  Hint: Use this link for info

e) Monitor the requests and responses between client and server.

- Explain how your Browser knows when to put up the login window (chrome://net-internals/#events)
- Explain how your server knows that you are logged in for subsequent request
- Copy the part that holds the Authorizations part and use the decode option on this link to comment what always must be done as a supplement when using Basic Http (and all other) authentication

f) Reflect (write down) the pros & cons of using Basic Authentication, and the use cases where it (still) could be relevant

2) Form-based Authentication (we will do this in the class)

a) Repeat step 1d+1e, but change the Authentication strategy from HTTP to Form-based Authentication. Use the same link as reference.

b) Limit the number of login tries:
Tomcat provides a special Realm LockOutRealm which we can use to extend other Realms (in our case the jdbcRealm we are using) with a user lock out mechanism if there are too many failed authentication attempts in a given period of time
Change the Realm declaration in Context.xml like this to test.

```
<Realm className="org.apache.catalina.realm.LockOutRealm" failureCount="2">
       <Realm className="org.apache.catalina.realm.JDBCRealm"
            .....
              "/>
</Realm>
```

Test by using an know user (peter_user@somewhere.dk) with a wrong password. You need to look at the NetBeans Tomcat log to see the “result”.

c) Reflect (write down) the pros & cons of using Form-based Authentication, and the use cases where it could be relevant

Authenticating API-endpoints (REST)

3) Basic/Form-based authentication with a REST-API

a) Repeat step **a)** from exercise-1, since we, one more time are leaving security to Tomcat.

b) Create, using the NetBeans Wizard, the skeleton code for a Restful web Service as you did last semester.
Don’t forget to add this to your pom-file:

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet</artifactId>
    <version>2.26</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>2.26</version>
</dependency>
```

c) Create the following two GET-endpoints:

- `xx/api/demo/user` : should return a simple json-string like: “Hello from user”
- `xx/api/demo/admin` : should return a simple json-string like: “Hello from user”

d) Secure the two endpoints, declaratively, similar to what you did in ex-1+2.

e) Test the two endpoints, using Postman (use Postmans Authorization tab to enter user and pw)

f) change the two endpoints to return the responses sketched below: (security principal + security context)

`xx/api/demo/user` : should return a simple json-string like: “Hello from USER: *users-email*”
`xx/api/demo/admin`: should return a simple json-string like: “Hello from ADMIN: *users-email*”

Hints: Read about the `SecurityContext` in section 16.1.1 in this chapter from the Jersey documentation. The `SecurityContext` and the `Principal` (which we get via `getUserPrincipal()`) interfaces are two important interfaces for Java Security, and in exercise 5 we shall see, that by creating our own implementations, we can create our own authentication strategy that will work, as the existing strategies

4) REST authentication with annotations

Repeat as above, but with annotations (as you did last semester) instead of web.xml
In order to make the security annotations work with Jersey, you must change add the `RolesAllowedDynamicFeature.class` in `ApplicationConfig` as sketched below (see jersey doc):

```
@Override
  public Set<Class<?>> getClasses() {
     Set<Class<?>> resources = new java.util.HashSet<>();
     resources.add(RolesAllowedDynamicFeature.class);
     addRestResourceClasses(resources);
     return resources;
  }
```

It seems (to me) a bit foolish, but to make the annotations work with jersey like this, you still need this section in web.xml (see this SO)

```
<security-constraint>
    <display-name>To Make Annotations Work</display-name>
    <web-resource-collection>
        <web-resource-name>All</web-resource-name>
        <description/>
        <url-pattern>/api/demo/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>user</role-name>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>jdbcRealm</realm-name>
</login-config>
```

Implementing our own JWT-based Authentication Strategy for a REST-API.

For the last 10-15 years, it seems like the Java Servlet API (and Java EE) has been stuck with Basic, Form-based, Digest and Client Certificate as alternatives you can use "out of the box". If this does not suit your needs, you will have to install your own packages or use alternative frameworks like Java's Spring Framework.
In the following, we will implement our own JWT-Authentication mechanism (obviously using an external package to handle the hardcore JWT operations).
The requirements for our solution will be:

- *After an initial traditional successful login, the client will be supplied with a valid JWT*
- *REST endpoints should be able to use the normal security annotations (see jersey doc)*
- *Access to endpoints will be determined by the validity/content of the supplied JWT*

a) Getting ready for the exercise:

Clone this project
https://github.com/SecurityDatFall2018/Week-6-Authentication.git
The project uses the same tables as given above but, uses JPA. Create the missing persistence.xml file and let it point to your existing database. Use the name "pu" so it can be used with the existing code (if your don't have created database, create it via the persistence-file).

The project continues, where the other exercises left, so check the following packages/classes:

- **The `rest` package**: Same as in exercise 3+4
- **The `exceptions` package**: Exceptions used by the example + a mapper to provide errors as JSON
- **The `entity` package**: Entity classes that maps to our previous DB. Observe the two methods: `verifyPassword(...)` and `getRolesAsString()` in the User entity-class. Also take a quick look at the simple `UserFacade` class in the package.
- **The `security` package**: This is where you need add code. Right now, it has an unfinished login-endpoint.
- Observe that the project does not include any security annotations.

This exercise will only focus on the backend, so we will use Postman as the client for all testing.

b) Test the POST-LoginEndpoint via Postman to verify you are connected to the database.

Add this JSON to the body to include username and password with the request:

```
{"username":"peter_user@somewhere.dk","password":"test"}
```

Also test with a non-existing user.

c) Test, and verify that we cannot access the two annotation-protected endpoints:

- `xx/api/demo/user`
- `xx/api/demo/admin`

The tasks in the following will be to 1) let the login endpoint return a valid token, and 2) allow access to protected endpoints for clients that supply a valid token (which will hold username and roles). You will be provided with all code in the following, but MAKE SURE you understand each step involved via the supplementing text.

d) *JWT tokens are not (normally) encrypted, but we need a shared key to use for the hash-algorithm when we:*

1) *Create the Token*
2) *Verify whether an incoming token was created by us*

In the `security` package create a new class **`SharedSecret`** and paste in the declaration given below (the dummy AAA.. key will make our life easier while we are implementing the solution):

```
/* This generates a secure random per execution of the server
 * A server restart, will generate a new key, making all existing tokens invalid
 * For production (and if a load-balancer is used) come up with a persistent key strategy */
public class SharedSecret {
    private static byte[] secret;
    public static byte[] getSharedKey() {
        System.out.println("****************** IMPORTANT *****************'");
        System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
        System.out.println("**** REMOVE FIXED SECRET BEFORE PRODUCTION ******");
        System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
        System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
        //REMOVE BEFORE PRODUCTION
        if(true){
            return "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA".getBytes();
        }

        if (secret == null) {
            secret = new byte[32];
            new SecureRandom().nextBytes(secret);
        }
        return secret;
    }
}
```

e) Next step is to create a token, including username, and roles (why is that “smart”) for an authenticated user after a successful login. Replace the dummy return value in the createToken(..) method (in the LoginEndpoint) with the content below.
Observe that all the complex JWT-operations are taken care of by the nimbus-jose-jwt package which is included by the pom-file

```
        StringBuilder res = new StringBuilder();
        for (String string : roles) {
            res.append(string);
            res.append(",");
        }
        String rolesAsString = res.length() > 0 ? res.substring(0, res.length() - 1) : "";
        String issuer = "semesterdemo_security_course";

        JWSSigner signer = new MACSigner(SharedSecret.getSharedKey());
        Date date = new Date();
        JWTClaimsSet claimsSet = new JWTClaimsSet.Builder()
                .subject(userName)
                .claim("username", userName)
                .claim("roles", rolesAsString)
                .claim("issuer", issuer)
                .issueTime(date)
                .expirationTime(new Date(date.getTime() + TOKEN_EXPIRE_TIME))
                .build();
        SignedJWT signedJWT = new SignedJWT(new JWSHeader(JWSAlgorithm.HS256), claimsSet);
        signedJWT.sign(signer);
        return signedJWT.serialize();
```

f) Test and verify that we get a Json Web Token (jwt) like you did in step b

g) Copy the token from the Postman request into the clipboard and paste it into this page: https://jwt.io/
Verify that we can read the content and answer the following questions:

- What is the advantages of having a Token with the provided information?
  - On the client?
  - On the server?
- Why is it not possible for hackers to create a similar Token, and use with our system?
- How should Tokens “always” be transported?

h) Implementing our own JWT-authentication strategy

Now we have reached the interesting part. We need to implement the required code that will make this work for our REST endpoints:

```
@RolesAllowed("user")
public String getFromUser(){
    String user = securityContext.getUserPrincipal().getName();
    return "\"Hello from USER: "+ user+"\"";
}
```

This will require the following steps to be performed:

1. Intercept the request before it reaches our endpoint-code (which we can do with a filter).
2. Check whether the request is for a protected resource (@RolesAllowed etc.)
3. If protected, check whether the request includes a valid token, (observe no need for a database lookup)
4. If token is not valid generate an error response (403)
5. If request was authenticated, generate a Principal and a SecurityContext instance to be user later in the request chain (when our rest endpoint code is called) for authorization

Note: All the following classes should be added to the `security` package

h1) Create a class `UserPrincipal` and paste in this code:

```
import entity.User;
import java.security.Principal;
public class UserPrincipal implements Principal {
  private String username;
  private List<String> roles = new ArrayList<>();

  /* Create a UserPrincipal, given the Entity class User*/
  public UserPrincipal(User user){
      this.username = user.getUserName();
      this.roles = user.getRolesAsStrings();
  }
  public UserPrincipal(String username, String... roles) {
    super();
    this.username = username;
    this.roles = Arrays.asList(roles);
  }
  @Override
  public String getName() {
    return username;
  }
  public boolean isUserInRole(String role) {
    return this.roles.contains(role);
  }
}
```

h2) Create a class `JWTSecurityContext` and paste in the following code:

```
import java.security.Principal;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.core.SecurityContext;
public class JWTSecurityContext implements SecurityContext {
    UserPrincipal user;
    ContainerRequestContext request;

    public JWTSecurityContext(UserPrincipal user,ContainerRequestContext request) {
        this.user = user;
        this.request = request;
    }
    @Override
    public boolean isUserInRole(String role) {
        return user.isUserInRole(role);
    }
    @Override
    public boolean isSecure() {
        return request.getUriInfo().getBaseUri().getScheme().equals("https");
    }
    @Override
    public Principal getUserPrincipal() {
        return user;
    }
    @Override
    public String getAuthenticationScheme() {
        return "JWT"; //Only for INFO
    }
}
```

h3)
Now it's time to write the filter that will intercept all requests and perform steps introduce in the beginning of step-f.

Open the JWTAuthenticationFilter.java file and insert the class given below.

This code is not as complex as you might think. Insert the lines below, the right places as comments to prove just that:

1. Intercept the request before it reaches our endpoint-code (which we can do with a filter).
2. Check whether the request is for a protected resource (@RolesAllowed etc.)
3. If protected, check whether the request includes a valid token, (observe no need for a database lookup)
4. If token is not valid generate an error response (403)
5. If request was authenticated, generate a Principal and a SecurityContext instance to be user later in the request chain (when our rest endpoint code is called) for authorization

```
@Provider
@Priority(Priorities.AUTHENTICATION)
public class JWTAuthenticationFilter implements ContainerRequestFilter {

  private static final List<Class<? extends Annotation>> securityAnnotations
          = Arrays.asList(DenyAll.class, PermitAll.class, RolesAllowed.class);
  @Context
  private ResourceInfo resourceInfo;

  @Override
  public void filter(ContainerRequestContext request) throws IOException {
    if (isSecuredResource()) {

      String token = request.getHeaderString("x-access-token");
      if (token == null) {
        request.abortWith(exceptions.GenericExceptionMapper.makeErrRes(token, 403));
        return;
      }
      try {
        UserPrincipal user = getUserPrincipalFromTokenIfValid(token);
        //What if the client had logged out????
        request.setSecurityContext(new JWTSecurityContext(user, request));
      } catch (AuthenticationException | ParseException | JOSEException ex) {
        Logger.getLogger(JWTAuthenticationFilter.class.getName()).log(Level.SEVERE, null, ex);
        request.abortWith(exceptions.GenericExceptionMapper.makeErrRes("Token not valid (timed
out?)", 403));
      }
    }
  }

  private boolean isSecuredResource() {

    for (Class<? extends Annotation> securityClass : securityAnnotations) {
      if (resourceInfo.getResourceMethod().isAnnotationPresent(securityClass)) {
        return true;
      }
    }
    for (Class<? extends Annotation> securityClass : securityAnnotations) {
      if (resourceInfo.getResourceClass().isAnnotationPresent(securityClass)) {
        return true;
      }
    }
    return false;
  }

  private UserPrincipal getUserPrincipalFromTokenIfValid(String token)
          throws ParseException, JOSEException, AuthenticationException {
    SignedJWT signedJWT = SignedJWT.parse(token);
    //Is it a valid token (generated with our shared key)
    JWSVerifier verifier = new MACVerifier(SharedSecret.getSharedKey());

    if (signedJWT.verify(verifier)) {
      if (new Date().getTime() > signedJWT.getJWTClaimsSet().getExpirationTime().getTime()) {
        throw new AuthenticationException("Your Token is no longer valid");
      }
      String roles = signedJWT.getJWTClaimsSet().getClaim("roles").toString();
      String username = signedJWT.getJWTClaimsSet().getClaim("username").toString();
      return new UserPrincipal(username, roles);
    } else {
      throw new JOSEException("User could not be extracted from token");
    }
  }
}
```

j) Were almost done. The only thing missing is to test the security functionality

j-1) Login, via postman as in step b) and copy the returned token into the clipboard.

j-2)
Test the two endpoints, as in step c)

Make sure to include the token with the request in a http-header “x-access-token”
Verify that it’s not enough to have a valid token, you must also have a valid role to be authorized.

Final reflections.

Answer the following questions:

- What is the advantage (if any) for a REST-based API of using JWT’s compared to session Cookies
- What is the advantage (if any) for a REST based API of using JWT’s compared to Basic HTTP Authentication
- What is the disadvantage (if any) with the implemented JWT-solution
- What will a client (Single Page WEB, Mobile App, etc.) have to do in order to use this API

Supplementing with OAuth2 and OpenID Connect

This is not an easy task with Java. Use this tutorial to create a working example (that allows you to authenticate) with a Google account (see the end of the article for hints about how to register with Google).
When this work, try to add it in to the example above, but first decide what should happen.

The example will allow you to get authenticated with a Google account, but first time the user authenticates we also want to “put him into our database”. The cool thing is that we know the email we get is real and belongs to a real client.