POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

DEI Departamento Engenharia Informática

Desenvolvimento de Aplicações Empresariais – 2021-22-1S

Engenharia Informática – 3.º ano – Ramo SI

Worksheet 7

Topics: Securing REST web services with JAX-RS and JSON Web Tokens (JWT)

How token-based authentication works (based on this post on stackoverflow)

In token-based authentication, the client exchanges *hard credentials* (such as username and password) for a piece of data called *token*. For each request, instead of sending the hard credentials, the client will send the token to the server to perform authentication and then authorization.

In a few words, an authentication scheme based on tokens follows these steps:

1. The client sends its credentials (username and password) to the server;
2. The server authenticates the credentials and, if they are valid, generates a token for the user;
3. The server stores the previously generated token in some storage along with the user identifier and an expiration date;
4. The server sends the generated token to the client;
5. The client sends the token to the server in each request;
6. The server, in each request, extracts the token from the incoming request. With the token, the server looks up the user details to perform authentication.
    - If the token is valid, the server accepts the request.
    - If the token is invalid, the server refuses the request.
7. Once the authentication has been performed, the server performs authorization;
8. The server can provide an endpoint to refresh tokens.

Note: Step 3 is not required if the server has issued a signed token (such as JWT, which allows you to perform *stateless* authentication).

Configuring a security domain in WildFly Java EE Server for token-based authentication

The new WildFly security framework is called Elytron (the Elytron subsystem) and allows for a single unified security framework across the whole application server.

To use a webserver that is configured already with Elytron to use JWT, please destroy the containers first (make down) re-clone and replace all the configuration files (Dockerfile, Makefile, scripts, docker-compose.yaml and the scripts folder).
Go to: https://github.com/dmp593/docker-wildfly-postgres.

The new changes were introduced since the commit 00d4597. Please view the commit to check what's changed and which files you need to replace.

Also, keep in mind that you need to re-match your .env file with the example: .env.example.
Your .env must also configure these environment variables:

```
… # keep the others!

JWT_ALGORITHM=RSA
JWT_ALIAS=alias
JWT_KEYSIZE=2048
JWT_KEYPASS=jwtsecret
JWT_KEYSTORE=jwt.keystore
JWT_STOREPASS=jwtsecret
JWT_CANONICAL_NAME=localhost
JWT_ORGANIZATIONAL_UNIT=estg
JWT_ORGANIZATION=ipl
JWT_COUNTRY=PT
JWT_ISSUER=quickstart-jwt-issuer
JWT_AUDIENCE=jwt-audience
JWT_REALM=jwt-realm
```

Configure your IntelliJ Java EE Academic Management project

After configuring a security domain in WildFly, we are now ready to add some Java EE code and configurations to our Academic Management project in IntelliJ. Code will include:

1. Hashing the user's password before storing it onto the database;
2. A `UserBean` to authenticate users;
3. A `Jwt` Java class to represent JWT token strings;
4. A JwtBean to generate JWT tokens;
5. A `LoginService` to expose REST web services to login and demonstrate token claims (token information about the username, role, validity, issuer, etc…);

1. Add the following `hashPassword(…)` method to the `User` entity, so that passwords are not stored in open text onto the database:

```
public static String hashPassword(String password) {
    char[] encoded = null;
    try {
        ByteBuffer passwdBuffer =
                Charset.defaultCharset().encode(CharBuffer.wrap(password));
        byte[] passwdBytes = passwdBuffer.array();
        MessageDigest mdEnc = MessageDigest.getInstance("SHA-256");
        mdEnc.update(passwdBytes, 0, password.toCharArray().length);
        encoded = new BigInteger(1, mdEnc.digest()).toString(16).toCharArray();
    } catch (NoSuchAlgorithmException ex) {
        Logger.getLogger(User.class.getName()).log(Level.SEVERE, null, ex);
    }
    return new String(encoded);
}
```

Note: this method uses a SHA-256 hash algorithm, which provides medium security strength. If you want to try other more secure (salted) algorithms, see, for instance, https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/

2. Change the password property initialization in the `User` entity constructor to:

```
this.password = hashPassword(password);
```

3. Create a new `UserBean` to authenticate users:

```
package ejbs;
import entities.User;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class UserBean {

    @PersistenceContext
    EntityManager em;
```

```
    public User authenticate(final String username, final String password) throws
Exception {
        User user = em.find(User.class, username);
        if (user != null &&
user.getPassword().equals(User.hashPassword(password))) {
            return user;
        }
        throw new Exception("Failed logging in with username '" + username + "':
unknown username or wrong password");
    }
}
```

4. Create a `Jwt` simple Java class to represent a JWT token string:

```
package jwt;

public class Jwt {
    private String token;

    public Jwt(String token) {
        this.token = token;
    }

    public void setToken(String token) {
        this.token = token;
    }

    public String getToken() {
        return token;
    }
}
```

We will now create a stateless `JwtBean` to generate and sign JWT tokens. We will use the JOSE library for this matter, as well as a JSON processing framework from Java EE. You can add these to your project before creating the JwtBean:

5. In the pom.xml, add a new dependency:

```
<dependencies>
  ...
  <dependency>
      <groupId>com.nimbusds</groupId>
      <artifactId>nimbus-jose-jwt</artifactId>
      <version>9.15.2</version>
  </dependency>
</dependencies>
```

And click on the button that reloads the maven dependencies (or run make build).

6. Create the `JwtBean` with the following code:

```
package ejbs;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
```

```
import java.security.Key;
import java.security.KeyStore;
import java.security.PrivateKey;

import javax.json.Json;
import javax.json.JsonArrayBuilder;
import javax.json.JsonObjectBuilder;

import com.nimbusds.jose.JOSEObjectType;
import com.nimbusds.jose.JWSAlgorithm;
import com.nimbusds.jose.JWSHeader;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.JWSSigner;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.RSASSASigner;

import javax.ejb.Stateless;

@Stateless(name = "JwtEJB")
public class JwtBean {
    static {
        FileInputStream fis = null;
        char[] password = "jwtsecret".toCharArray();
        String alias = "alias";
        PrivateKey pk = null;
        try {
            KeyStore ks = KeyStore.getInstance("JKS");
            String configDir = System.getProperty("jboss.server.config.dir");

            // check .env
            String keystorePath = configDir + File.separator + "jwt.keystore";
            fis = new FileInputStream(keystorePath);
            ks.load(fis, password);
            Key key = ks.getKey(alias, password);
            if (key instanceof PrivateKey) {
                pk = (PrivateKey) key;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {}
            }
        }
        privateKey = pk;
    }

    private static final PrivateKey privateKey;
    private static final int TOKEN_VALIDITY = 14400;
    private static final String CLAIM_ROLES = "groups";
    private static final String ISSUER = "quickstart-jwt-issuer"; // check .env
    private static final String AUDIENCE = "jwt-audience"; // check .env

    public String createJwt(final String subject, final String[] roles) throws
Exception {
        JWSSigner signer = new RSASSASigner(privateKey);
        JsonArrayBuilder rolesBuilder = Json.createArrayBuilder();
        for (String role : roles) { rolesBuilder.add(role); }

        JsonObjectBuilder claimsBuilder = Json.createObjectBuilder()
                .add("sub", subject)
```

```
                .add("iss", ISSUER)
                .add("aud", AUDIENCE)
                .add(CLAIM_ROLES, rolesBuilder.build())
                .add("exp", ((System.currentTimeMillis() / 1000) +
TOKEN_VALIDITY));

        JWSObject jwsObject = new JWSObject(new
JWSHeader.Builder(JWSAlgorithm.RS256)
                .type(new JOSEObjectType("jwt")).build(),
                new Payload(claimsBuilder.build().toString()));

        jwsObject.sign(signer);

        return jwsObject.serialize();
    }
}
```

Briefly, this `JwtBean` gets the private key from the previously created keystore in step 2 to create and sign a JWT with the authenticated username, role and a validity.

7. Let's now create the `LoginService` in the `ws` package with the `authenticateUser(…)` and `demonstrateClaims(…)` REST web services:

```
package ws;

import com.nimbusds.jwt.JWT;
import com.nimbusds.jwt.JWTParser;

import ejbs.UserBean;
import entities.User;
import jwt.Jwt;
import ejbs.JwtBean;

import javax.ejb.EJB;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import java.text.ParseException;
import java.util.logging.Logger;
@Path("/auth")
public class LoginService {

    private static final Logger log =
Logger.getLogger(LoginService.class.getName());

    @EJB
    JwtBean jwtBean;

    @EJB
    UserBean userBean;

    @POST
    @Path("/login")
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public Response authenticateUser(@FormParam("username") String username,
                                     @FormParam("password") String password) {

        try {
            User user = userBean.authenticate(username, password);
```

```
            if (user != null) {
                if (user.getUsername() != null) {
                    log.info("Generating JWT for user " + user.getUsername());
                }
                String token = jwtBean.createJwt(user.getUsername(), new
String[]{user.getClass().getSimpleName()});
                return Response.ok(new Jwt(token)).build();
            }
        } catch (Exception e) {
            log.info(e.getMessage());
        }
        return Response.status(Response.Status.UNAUTHORIZED).build();
    }

    @GET
    @Path("/user")
    public Response demonstrateClaims(@HeaderParam("Authorization") String auth) {
        if (auth != null && auth.startsWith("Bearer ")) {
            try {
                JWT j = JWTParser.parse(auth.substring(7));
                return Response.ok(j.getJWTClaimsSet().getClaims()).build();
//Note: nimbusds converts token expiration time to milliseconds
            } catch (ParseException e) {
                log.warning(e.toString());
                return Response.status(400).build();
            }
        }
        return Response.status(204).build(); //no jwt means no claims to extract
    }
}
```

Briefly, this component exposes 2 web service methods under the `/auth` URL path: 1) the `authenticateUser(…)` POST method at the `/login` URL receives a username and password payload in a FORM_URLENCONDED type, authenticates the user and produces a JWT token in JSON; and 2) the `demonstrateClaims(…)` GET method at `/user` receives a JWT token and shows information on its username, role, validity, etc.

Configuring security constraints/permissions on our Java EE REST service layer

In Java EE applications, you can secure your REST web services using one of the following methods:

1. Updating the `web.xml` deployment descriptor to define security configuration;
2. Applying annotations to your JAX-RS classes;
3. Using the `javax.ws.rs.core.SecurityContext` interface to implement security programmatically.

For example, let's configure security constraints/permissions in a `web.xml` deployment descriptor of our Academic Management Java EE application, so that:

- Administrators and Teachers can access the students list;
- Only Administrators can create, update or delete students;

8. For this purpose, **create a `web.xml` file at `src/main/webapp/WEB-INF`**, and paste:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

   <security-constraint>
        <web-resource-collection>
            <web-resource-name>Students List</web-resource-name>
            <url-pattern>/api/students</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>Administrator</role-name>
            <role-name>Teacher</role-name>
        </auth-constraint>
    </security-constraint>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Students create</web-resource-name>
            <url-pattern>/api/students</url-pattern>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>Administrator</role-name>
        </auth-constraint>
    </security-constraint>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Students update and delete</web-resource-name>
            <url-pattern>/api/students/*</url-pattern>
            <http-method>PUT</http-method>
            <http-method>DELETE</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>Administrator</role-name>
        </auth-constraint>
    </security-constraint>
    <security-role>
        <description>Administrator role</description>
        <role-name>Administrator</role-name>
    </security-role>
    <security-role>
        <description>Teacher role</description>
        <role-name>Teacher</role-name>
    </security-role>
    <login-config>
        <auth-method>BEARER_TOKEN</auth-method>
        <realm-name>jwt-realm</realm-name>
    </login-config>
</web-app>
```

Each security constraint in the web.xml file refers to the collection of web resources to secure and the roles allowed for accessing those resources. Web resources can be expressed by unique URL pattern mappings (for instance `/api/students`) or extended path mappings with a "*" (for instance `/api/students/*`). We can also opt to specify which HTTP methods to allow, as you can observe in the example above (if not specified, all HTTP methods are allowed).

The `<login-config>` tag contains the authentication method to use (in our case, BEARER_TOKEN), as well as the security realm where JWT tokens get validated. The referenced `jwt-realm` is the realm we previously configured in WildFly.

Now, let's run our Academics Management Java EE project and do some HTTP rest requests to test it.

9. Save and re-run the application (make monitor).

10. Add the following HTTP request methods to your REST test `.http` file:

```
### Get JWT token
POST http://localhost:8080/academics/api/auth/login
Content-Type: application/x-www-form-urlencoded

username=replace this with a valid student/teacher/administrator
username&password=replace this with a valid student/teacher/administrator password


### Who am I?
GET http://localhost:8080/academics/api/auth/user
Accept: application/json
Authorization: Bearer replace this with generated JWT token

##### Get students list
GET http://localhost:8080/academics/api/students
Accept: application/json
Authorization: Bearer replace this with generated JWT token

### Create new student
POST http://localhost:8080/academics/api/students
Content-Type: application/json
Authorization: Bearer replace this with generated JWT token

{ "email": "jane@mail.com",
  "name": "Jane Doe",
  "username": "1234567",
  "password": "aaaaa",
  "courseCode":"1",
  "courseName": "EI"
}

### Enroll student in a subject - run after the previous POST
PUT http://localhost:8080/academics/api/students/1234567/subjects/enroll/1
Authorization: Bearer replace this with generated JWT token

### Get a student's details
GET http://localhost:8080/academics/api/students/1111111
Accept: application/json
Authorization: Bearer replace this with generated JWT token

### Delete a student - run after the previous POST
DELETE http://localhost:8080/academics/api/students/1234567
Accept: application/json
Authorization: Bearer replace this with generated JWT token
###
```

11. Execute the first `### Get JWT token` HTTP request and verify that a JWT token for a Teacher t1 is shown in the Response. Copy that JWT token string (without the "") and paste it

after the `Authorization: Bearer` string for the next HTTP requests. Execute those next requests and verify the results.

Hint: you can generate right away tokens for an Administrator and a Student as well, and save all of them in a text file, since they will be valid for the next 14400 seconds = 4 hours. It will then be easier for you to test permissions to your REST web methods for these different roles.

Let's now assume that only Administrators and Students can get a list of Courses. We will use Java EE annotations for this purpose.

Note: this is an alternative way of establishing authorization permissions in Java EE. Mixing security annotations with those established in web.xml is messy. Be careful, since clashing permissions will be overridden by those in the web.xml descriptor.

12. Add the following context parameter to your web.xml file (right before the first <security-contraint> element), in order to enable role-based security in WildFly's default REST library RESTEasy:

```
…
    <context-param>
        <param-name>resteasy.role.based.security</param-name>
        <param-value>true</param-value>
    </context-param>
…
```

13. Add the `@RolesAllowed` annotation on top of the `getAllCourses()` method on the `CourseService`, like this:

```
@GET
@Path("/")
@RolesAllowed({"Administrator","Student"})
public List<CourseDTO> all(){…}
```

14. Re-run and test this new declarative security permission with the appropriate HTTP requests (do not forget to interchange JWT tokens to verify all permissions);
Note: declarative security includes also the @DeclareRoles, @PermitAll and @DenyAll annotations. The @DeclareRoles purpose is to enumerate which roles can be used to access a certain class's methods (this goes usually at the top of a class declaration), as the @PermitAll and @DenyAll annotations allow for access to all or none of authenticated users trying to access a certain method.

Now, we will provide access to student's details to all Administrators and Teachers, but only to a student whose username is the same of the one being accessed (*as a student, I can only see my details*). Since web.xml url-patterns do not support path parameters (such as {username}), and declarative security goes only on top of methods, we will use programmatic security for this purpose.

15. Inject the SecurityContext context in your StudentService class as this:

```
@Context
private SecurityContext securityContext;
```

Note: you could also inject this security context directly through the `getStudentDetails(…)` method parameters, but we may need it for other methods in the future.

16. Head to the `getStudentDetails(…)` web method in your `StudentService` component, and make it similar to the following code:

```
@GET
    @Path("{username}")
    public Response getStudentDetails(@PathParam("username") String username)
            throws MyEntityNotFoundException {

        Principal principal = securityContext.getUserPrincipal();
        if(!(securityContext.isUserInRole("Administrator") ||
                securityContext.isUserInRole("Teacher") ||
                securityContext.isUserInRole("Student") &&
principal.getName().equals(username))) {
            return Response.status(Response.Status.FORBIDDEN).build();
        }

        Student student = studentBean.findStudent(username);
        if(student == null) {
            throw new MyEntityNotFoundException("Student with username " +
username + " not found.");
        }

        return Response.status(Response.Status.OK)
                 .entity(toDTO(student))
                .build();
    }
```

17. Re-run your application and perform the necessary security tests executing HTTP requests with different tokens (try to access a student's details with that student username, with another student username, with a Teacher and with an Administrator).

18. Apply other security permissions using one of the above security approaches (web.xml, annotations or through the security context methods).

Bibliography

Java EE Tutorial 8 (all of it).