Lecture 4: Security and Access Control

TIES4560 SOA and Cloud Computing
Autumn 2023

University of Jyväskylä

Khriyenko Oleksiy

REST Web Services

Message Body Writer *is used to convert various objects to correspondent type of a Response body.*

- *By default, there are imbedded Message Body Writers that Jersey uses to convert objects to some of the formats of Response body.*
- *If you need to convert some type (as well as custom one) to a required type of Response body, you have to implement generic* ***MessageBodyWriter*** *interface as a provider.*
- *Provide annotation of the Media Type your writer produces. You can have different implementations of writer for different types.*

```
…
@Provider
@Produces(MediaType.TEXT_PLAIN)
public class DateMessageBodyWriter implements MessageBodyWriter<Date>{
@Override
public long getSize(Date arg0, Class<?> arg1, Type arg2, Annotation[] arg3, MediaType arg4)
{// is deprecated in JAX-RS 2.0, we do not implement it, recommended value to return is -1.
   return -1;
}
@Override
public boolean isWriteable(Class<?> type, Type arg1, Annotation[] arg2, MediaType arg3) {
// return true if this writer supports particular type (here we do this for Date type)...
   return Date.class.isAssignableFrom(type);
}
@Override
public void writeTo(Date date, Class<?> type, Type type1, Annotation[] annot, MediaType mt,
MultivaluedMap<String, Object> mm, OutputStream out) throws IOException,
                                                          WebApplicationException {
   out.write(date.toString().getBytes());
}
}
```

REST Web Services

Custom Media Types *are acceptable as long as you support them with corresponding Message Body Writers and Readers.*

```
@GET
@Produces(value={MediaType.TEXT_PLAIN, "text/customDate"})
@Path("/dueDate")
public Date getDueDate() throws ParseException{
   SimpleDateFormat sdf = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
   String dateInString = "31-01-2017 23:59:59";
   Date dueDate = sdf.parse(dateInString);
   return dueDate;
}
```

```
@Provider
@Produces("text/customDate")
public class CustomDateMessageBodyWriter implements MessageBodyWriter<Date>{
//two first methods are the same as in previouse Writer
@Override
   public void writeTo(Date date, Class<?> type, Type type1, Annotation[] annot, MediaType
mt, MultivaluedMap<String, Object> mm, OutputStream out) throws IOException,
                                                                WebApplicationException {
      String s="th";
      String dayS = new SimpleDateFormat("DD").format(date);
      if(dayS.equals("1"))s="st";
      if(dayS.equals("2"))s="nd";
      if(dayS.equals("3"))s="rd";
      String monthS = new SimpleDateFormat("MMMM").format(date);
      String yearS = new SimpleDateFormat("YYYY").format(date);
      String customDate = ""+dayS+""+s+" "+monthS+" "+yearS;
      out.write(customDate.getBytes());
   }
}
```

REST Web Services

*JAX-RS has a **Client** object to support REST API calls.*

- *It is an interface, and we have to use a **ClientBuilder** to get the instance of the object.*
- *Specify a target REST API to point the client, build a request, invoke a http method, convert to the required type.*

```
public class RESTwsClient {
   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      Response response = client.target("http://.../webapi/publications/1").request().get();
      Publication publ = response.readEntity(Publication.class);
      System.out.println(publ.getTitle()); }
}
```

```
public class RESTwsClient {
   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      Publication publ = client.target("http://localhost:8080/RESTws/webapi/publications/1")
                              .request(MediaType.APPLICATION_JSON)
                              .get(Publication.class);
      System.out.println(publ.getTitle()); }
}
```

```
public class RESTwsClient {
   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      String response = client.target("http://localhost:8080/RESTws/webapi/publications/1")
                             .request(MediaType.APPLICATION_JSON)
                             .get(String.class);
      System.out.println(response);   }
}
```

REST Web Services

*JAX-RS has a **Client** object to support REST API calls.*

- *It is a good practice to operate with customized targets of the client to address different resources.*
- *Doing a POST request make an object conversion to corresponding representation (content type) of the request entity.*
- *Use a Response instance to get an access to a bunch of useful metadata.*

```
public class RESTwsClient {
   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      WebTarget baseTarget = client.target("http://localhost:8080/RESTws/webapi/");
      WebTarget publicationsTarget = baseTarget.path("publications");
      WebTarget publicationTarget = publicationsTarget.path("{publicationId}");
      WebTarget profilesTarget = baseTarget.path("profiles");
      WebTarget profileTarget = profilesTarget.path("{profileName}");

      Response getResponse = publicationTarget.resolveTemplate("publicationId", "1")
                                              .request(MediaType.APPLICATION_JSON)
                                              .get();
      Publication publication = getResponse.readEntity(Publication.class);
      System.out.println(publication.getTitle());

      Profile newProfile = new Profile("me", "Oleksiy", "Khriyenko");
      Response postResponse = profilesTarget.request().post(Entity.json(newProfile));
      if(postResponse.getStatus() != 201){System.out.println("Error: Profile is not created.");
      }else {
         Profile respProfile = postResponse.readEntity(Profile.class);
         System.out.println(respProfile.getProfileName()+": "+respProfile.getFirstName()+" "
                                                              +respProfile.getLastName());
      }
   }
}
```

REST Web Services

JAX-RS has an ***Invocation*** *mechanism to outsource Request preparation to a separate method(s).*

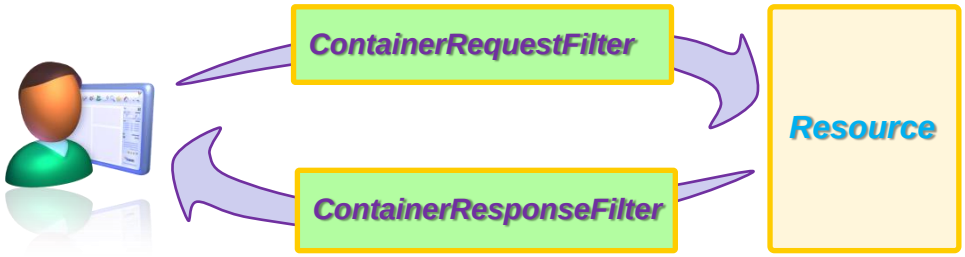
```
public class InvocationOption {

   public static void main(String[] args) {

      InvocationOption option = new InvocationOption();
      Invocation invocation = option.prepareRequestForPublicationByYear(2022);
      Response invocResp = invocation.invoke();
      if(invocResp.getStatus() == 200){
         System.out.println(invocResp.readEntity(String.class));
      }else{
         System.out.println("Error!!!");
      }
   }

   private Invocation prepareRequestForPublicationByYear(int year) {

      Client client = ClientBuilder.newClient();
      return client.target("http://localhost:8080/RESTws/webapi/publications")
                   .queryParam("year", year)
                   .request(MediaType.APPLICATION_JSON)
                   .buildGet();
   }
}
```

REST Web Services

Filters *in JAX-RS*

- *Used to apply some common logic/action for many APIs and help to avoid multiple repetitions.*
  - **ContainerRequestFilter** *implements a filter that is called before the Request is about to be applied to the API it is meant for.*
  - **ContainerResponseFilter** *implements a filter that is called before the Response is about to be sent. So, we have a possibility to modify a value of the Response header.*

Depending on your goal you may use Request or Response filter…



REST Web Services

Filters *in JAX-RS*

- *put specific label to the service (e.g.* ***X-Powered-By TIES-4560****). Create a class that implements ContainerResponseFilter class and make necessary modifications of the Response header.*

```
…
@Provider
public class PoweredByResponseFilter implements ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
                                                        responseContext) throws IOException {
        responseContext.getHeaders().add("X-Powered-By", "TIES-4560");
    }
}
```

- *to perform a logging of the header metadata before and after a resource (API) is used. Create a class that implements both ContainerRequestFilter and ContainerResponseFilter classes.*

```
…
@Provider
public class LoggingFilter implements ContainerRequestFilter, ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        System.out.println("Request Headers: " + requestContext.getHeaders());
    }
    @Override
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
                                                        responseContext) throws IOException {
        System.out.println("Response Headers: " + responseContext.getHeaders());
    }
}
```

REST Web Services

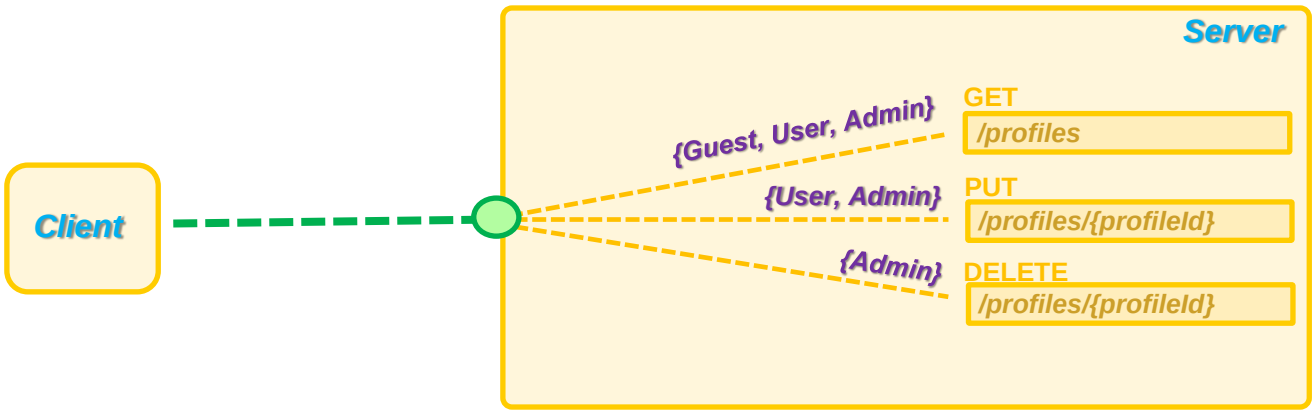
ACCESS CONTROL *of REST Web Services*

AUTHENTICATION

User Identity

AUTHORIZATION

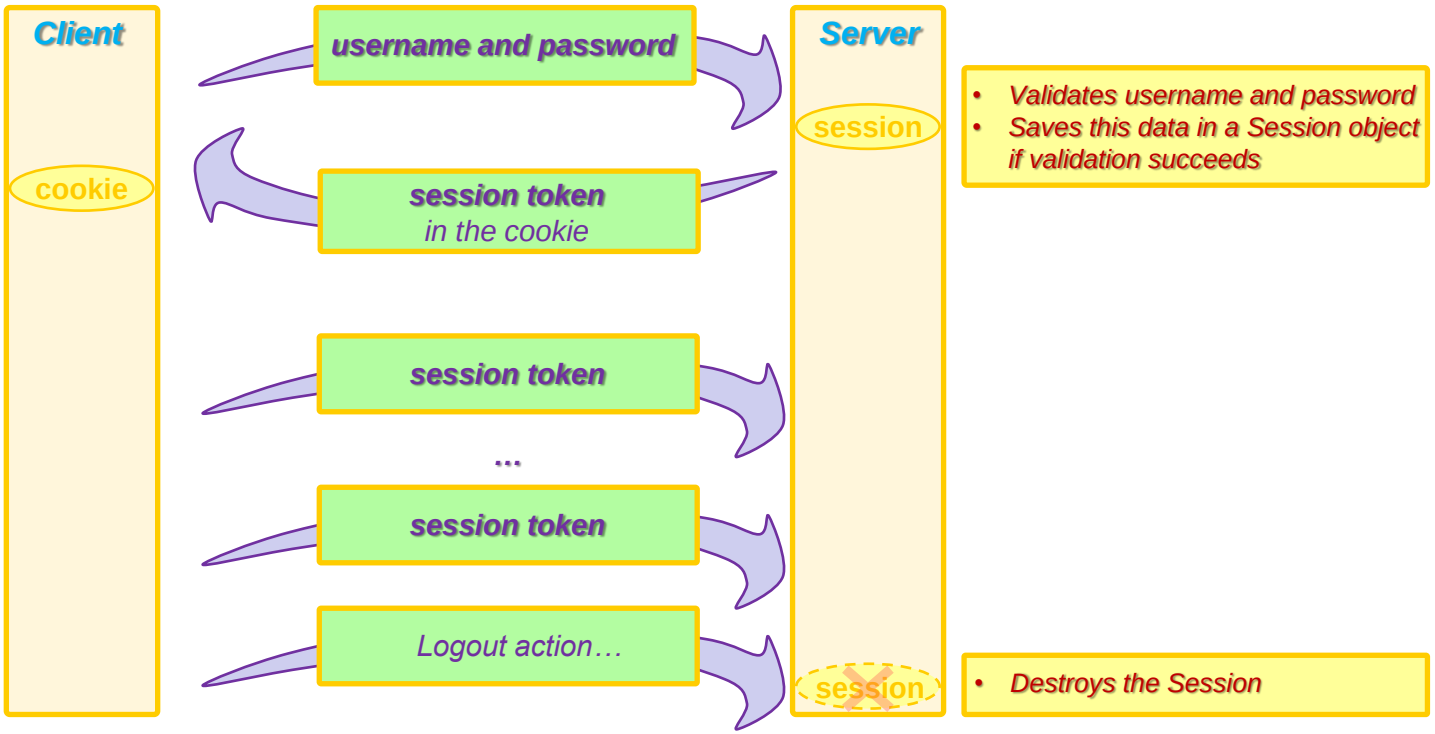
User Permission



REST Web Services

AUTHENTICATION

- *Classic Web Application authentication based on **sessions**.*



*… but this is not applicable for REST API, since it supposed to be **stateless**!!!*

REST Web Services

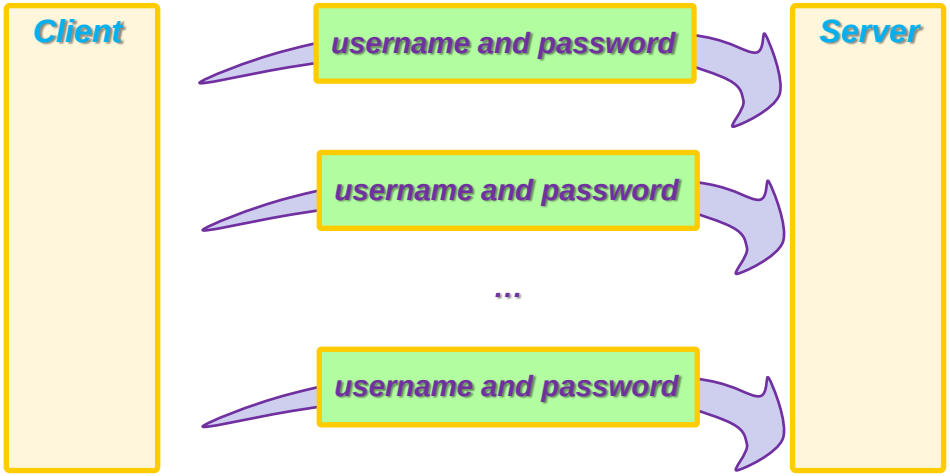
AUTHENTICATION

- *Basic Access Authentication (https://en.wikipedia.org/wiki/Basic_access_authentication)*
- *Digest Access Authentication (https://en.wikipedia.org/wiki/Digest_access_authentication)*
- *Asymmetric (Public-key) Cryptography (https://en.wikipedia.org/wiki/Public-key_cryptography)*
- *OAuth 1.0 and OAauth 2.0(https://en.wikipedia.org/wiki/OAuth)*
- *JSON Web Tokens (JWT) (https://en.wikipedia.org/wiki/JSON_Web_Token)(https://jwt.io/introduction/)*
- *Hawk Authentication (https://blog.notmyidea.org/whats-hawk-and-how-to-use-it.html) (https://blog.mozilla.org/services/2015/02/05/whats-hawk-and-how-to-use-it/)*
- *AWS Signature (http://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html#auth-methods-intro)*

REST Web Services

AUTHENTICATION

Basic Auth *(Basic Access Authentication) – the most basic mechanism for REST APIs.*



- *The username and password is sent in the* ***Request header*** *as a string value in* ***username:password*** *form.*
- *The mechanism does not assume any encryption, but it uses* ***Base64 encoding*** *of that string to avoid any problem with non-HTTP-compatible characters.*
- *Finally, the username and password are added to the Request header as a value of* ***Authorization*** *key in a form: "****Basic*** *<the encoded string>".*

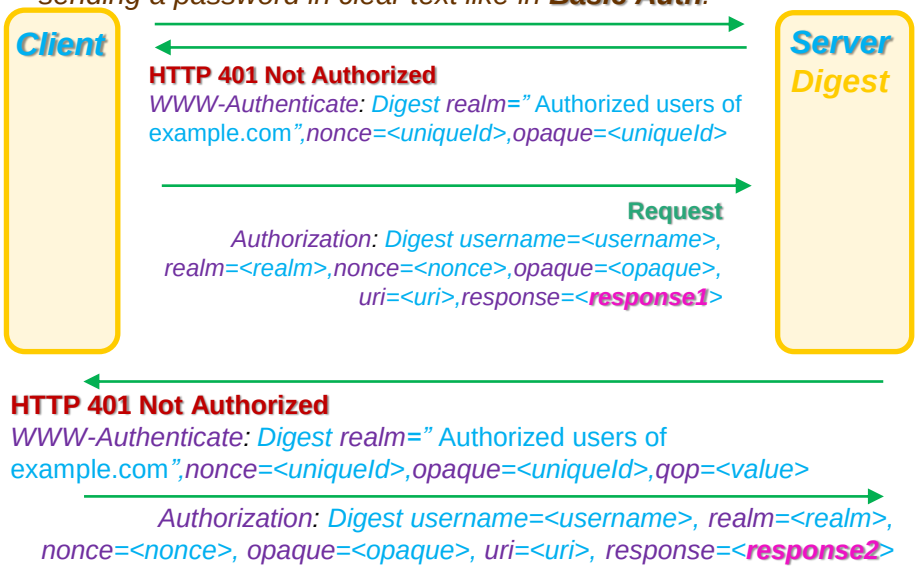
```
GET /resource HTTP/1.1 Host: server.example.com
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

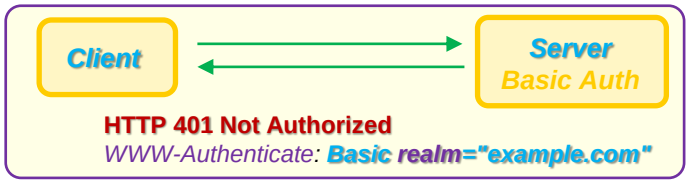
… there is no security in the mechanism.
To avoid access to the header data, ***send request over HTTPS****!!!*

REST Web Services

AUTHENTICATION

Digest Access Authentication – *method to send credentials using a combination of the password and other bits of information to create an MD5 hash which is then sent to the server to authenticate. Sending a hash avoids the problems with sending a password in clear text like in* ***Basic Auth****.*





nonce *– generated by Server value unique for each request. It should be Base64 encoded or hexadecimal (it also could be hashed). It will be used by Client to generate a hash to send back to the Server.*

opaque *– another unique generated by Server value. It should be Base64 encoded or hexadecimal (it also could be hashed). It will be returned by Client unaltered.*

realm *- just a string to display to users so they know which username and password they should provide (e.g. "Authorized users of example.com"). It's also used by the client to generate a hash to send back to the server.*

qop *- quality of protection, can have a value of "auth" (by default) or "auth-int". "auth" is used for client authentication only (initial standard). "auth-int" - an attempt to provide some level of integrity protection of the response as well and the client must also include the request body as part of the message digest.*

cnonce *– (client nonce) is similar to nonce but is generated by the client. The cnonce figures into the response digest computed by the client and its original value is passed along to the server so that it can be used there to compare digests. This provides some response integrity and mutual authentication, in that both the client and server have a way to prove they know a shared secret. When the qop directive is sent by the server, the client must include a cnonce value.*

nc *– (nonce count) is a hexadecimal count of the number of requests that the client has sent with a given nonce value. This way, the server can guard against replay attacks.*

response1 **=**

MD5(MD5("username:realm:password") :nonce: MD5("requestMethod.requestURI"))

response2 **=**

MD5(MD5("username:realm:password") :nonce:nc:cnonce:qop: MD5("requestMethod.requestURI". MD5(<responseBody>)))

auth / auth-int

Advantages:

- *passes a hashed value instead of text;*
- ***nonce*** *drastically changes the computed hash on each new request;*
- ***nc*** *value is helpful at preventing replay attacks;*

Weaknesses:

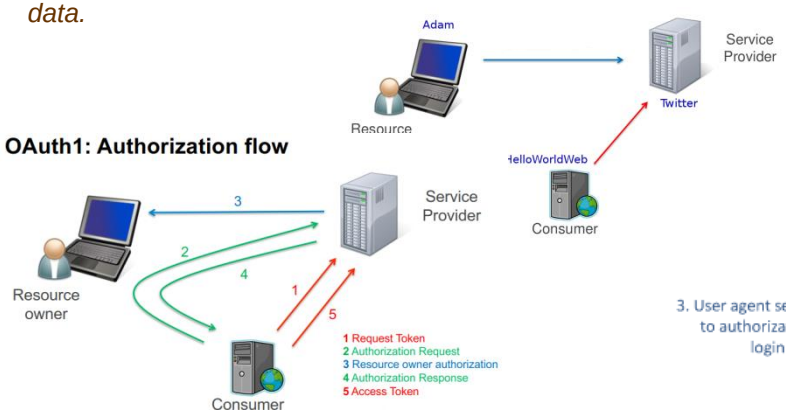
- *Enhancements were optional and might be not implemented;*
- ***MD5*** *is not a strong hashing algorithm (Bcrypt is preferable);*
- *There's also no way for a server to verify the identity of the requesting client ;*

REST Web Services

AUTHENTICATION

OAuth *– is a specification that defines secure authentication model on behalf of another user (https://oauth.net/). There are two versions (OAuth 1 and OAuth 2). OAuth 2 is the latest version, and it is not backward compatible with OAuth 1.*

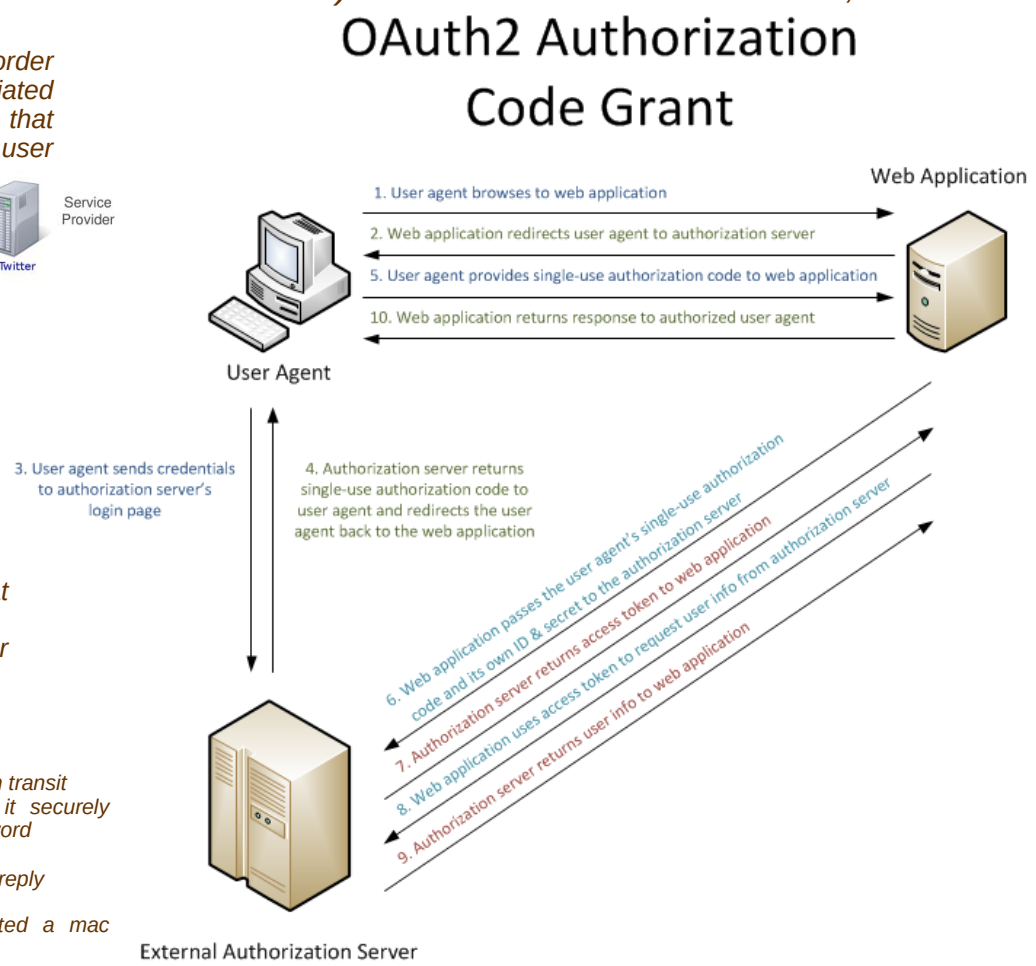
- *OAuth is widely used in popular social Web sites in order to grant access to a user account and associated resources for a third-party consumer (application) that usually uses RESTful Web Services to access the user data.*



- ***OAuth 1.0*** *uses signature in authorization header that does not require SSL (complex for implementation).*
- ***OAuth 2.0*** *should use SSL to be secured, has four Grant types and two Access Token Types:*
  - ***Authorization Code Grant***
  - ***Implicit Grant*** *(e.g. JavaScript client)*
  - ***Resource Owner Password Credentials Grant***
  - ***Client Credentials Grant*** *(client app authentication)*

Bearer
- *Large random token*
- *Needs SSL to protect it in transit*
- *Server needs to store it securely hashed like a user password*

Mac
- *Uses a nonce to prevent reply*
- *Does not require SSL*
- *OAuth 1.0 only supported a mac type token*



REST Web Services

AUTHENTICATION

JWT *(JSON Web Tokens, pronounced "jot") – basic mechanism based on tokens that contain information that is unique to a user, but may also contain any additional information that the user may need.*

- *All the information is transferred within JSON message*
- *It is not 100% secured, but it contains a digital signature that is cryptographically encrypted using a strong algorithm such as HMAC SHA-256. There is another standard for encrypted content - JSON Web Encryption (JWE)*
- *It is supported by variety of programming languages (Java, JS, Node.js, .NET, Python, PHP, Ruby, Go, etc.)*
- *Could be sent as a part of URL (Query string), Form body parameter, cookie or HTTP Header (x-access-token)*
- *Is used also in **OpenID Connect** (Authentication via third party)*

Structure

Header

```
Base64encode (
{
  "typ": "JWT",
  "alg": "HS256"
}
)
```

Payload

```
Base64encode (
{
   "iss": "TIES456_RESTws",
   "iat": 1475158043,
   "exp": 1506694043,
   "aud": "www.client.com",
   "sub": "jrocket@client.com",
   "GivenName": "Johnny",
   "Surname": "Rocket",
   "Email": "jrocket@client.com",
   "Role": [
      "Manager",
      "Project Administrator"
   ]
}
)
```

*The payload contains the main part – **JWT claims**. These claims are broken up into **registered** claims, **public** claims and **private** claims.*

- *JWT Builder (http://jwtbuilder.jamiekurtz.com)*
- *JWT Verifier (http://jwt.io)*
- *Base64 En/Decoder (https://www.base64decode.org)*

Signature

```
Base64encode (
   HMAC-SHA56 (
      Base64encode(header) + ”.” +
      Base64encode(payload) ,
      Private secrete key
   )
)
```

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJUSUVTNDU2X1JFU1R3cyIsImlhdCI6MTQ3NTE1ODA0MywiZXhwIjoxNTA2Njk0MDQzLCJhdWQiOiJ3d3cuY2xpZW50LmNvbSIsInN1YiI6Impyb2NrZXRAY2xpZW50LmNvbSIsIkdpdmVuTmFtZSI6IkpvaG5ueSIsIlN1cm5hbWUiOiJSb2NrZXQiLCJFbWFpbCI6Impyb2NrZXRAY2xpZW50LmNvbSIsIlJvbGUiOlsiTWFuYWdlciIsIlByb2plY3QgQWRtaW5pc3RyYXRvciJdfQ.O07YPk23arTI3htjrJaIaBH4TaN5VQmR0HQhETr1UV8

REST Web Services

AUTHENTICATION

- *by Server, **JWT** could be sent in the body of the Response or in the Response header, as a cookie…*
- *by Client, **JWT** is sent in the Request header as a value of **Authorization** key in a form: "**Bearer** <token>".*

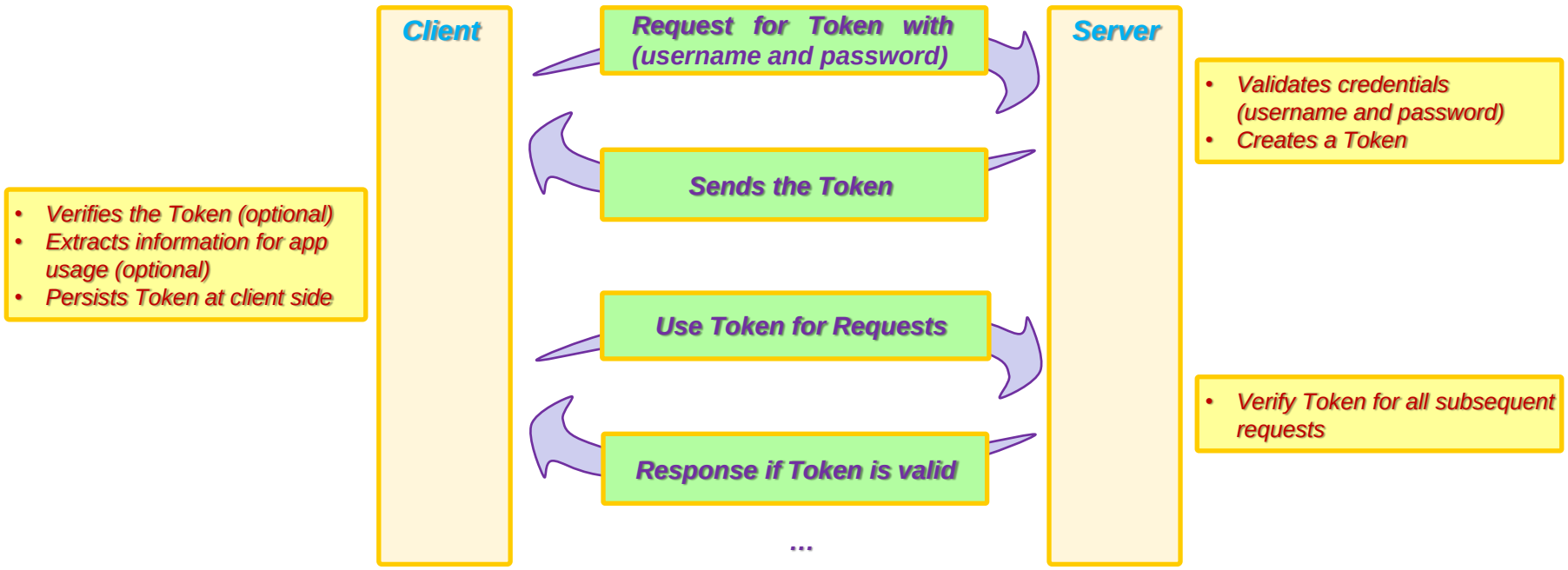
```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

```
…
{ "access_token":"your.jwt.here",
  "token_type":"JWT",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

```
GET /resource HTTP/1.1 Host: server.example.com
Authorization: Bearer your.jwt.here
```

```
…
{ "code":"401",
  "error ":"invalid_token ",
  "error_description ":"The access token expired"
}
```

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example",
        error="invalid_token",
        error_description="The access token expired"
```



REST Web Services

AUTHENTICATION and AUTHORIZATION

You can:

- *Delegate simple authentication and authorization to your container (Tomcat) or to your frontend (Apache). The protected resources, roles and access areas should be configured at container level in web.xml.*

```
…
<security-constraint>
   <web-resource-collection>
      <url-pattern>/*</url-pattern>
      <http-method>POST</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
```

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>/profiles/*</url-pattern>
      <http-method>GET</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>admin</role-name>
      <role-name>user</role-name>
      <role-name>guest</role-name>
</auth-constraint>
</security-constraint>
```

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>/profiles/*</url-pattern>
      <http-method>PUT</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>my-realm</realm-name>
</login-config>
```

- *Delegate authentication to your container or frontend and let Jersey manage authorization with JSR-250 annotations such as @PermitAll, @DenyAll, @RolesAllowed and specified SecurityContex.*

- *Be independent from any container and let "Jersey-based" application take care of authentication and authorization by implementing own ContainerRequestFilter.*

- *Let Spring Security manage authentication and authorization.*

REST Web Services

AUTHENTICATION *in JAX-RS could be organized via filtering of Request header.*

Basic Auth

```
…
@Provider
public class SecurityFilter implements ContainerRequestFilter {
   private static final String AUTHORIZATION_HEADER_KEY = "Authorization";
   private static final String AUTHORIZATION_HEADER_PREFIX = "Basic ";
   private static final String SECURED_URL_PREFIX = "secured";
   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException {
      if ((requestContext.getUriInfo().getPath().contains(SECURED_URL_PREFIX))
        ||(requestContext.getMethod().equals("DELETE"))) {
         List<String> authHeader = requestContext.getHeaders().get(AUTHORIZATION_HEADER_KEY);
         if (authHeader != null && authHeader.size() > 0) {
            String authToken = authHeader.get(0);
            authToken = authToken.replaceFirst(AUTHORIZATION_HEADER_PREFIX, "");
            String decodedString = Base64.decodeAsString(authToken);
            StringTokenizer tokenizer = new StringTokenizer(decodedString, ":");
            String username = tokenizer.nextToken();
            String password = tokenizer.nextToken();
            if ("user".equals(username) && "password".equals(password)) { return; }
         }
         ErrorMessage errorMessage = new ErrorMessage("User cannot access the resource.", 401,
                                                      "http://myDocs.org");
         Response unauthorizedStatus = Response.status(Response.Status.UNAUTHORIZED)
                                               .entity(errorMessage)
                                               .build();
         requestContext.abortWith(unauthorizedStatus);
      }
   }
}
```

REST Web Services

AUTHORIZATION in JAX-RS could be organized via *SecurityContext*

- *We can inject SecurityContext into resource and make various verifications inside the methods...*

```
…
@Path("/profiles")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ProfileResource {

            private ProfileService profileService = new ProfileService();
            @Context
            private SecurityContext securityContext;

            @GET
            @Path("/{profileName}")
            public Profile getProfile(@PathParam("profileName") String profileName){
                        if (!securityContext.isUserInRole("admin")){
                                    throw new WebApplicationException("Not authorized", 401);
                        }
                        return profileService.getProfile(profileName);
            }
            …
}
```

You may use SecurityContext as well as a parameter of the method (then it will be available only in that method)…

- *Now we have to **customize Security Context** by implementing SecurityContext interface.*
- *Finally, we have to add our SecurityContext to the RequestContext of our SecurityFilter.*

REST Web Services

AUTHORIZATION *in JAX-RS*

- *customize SecurityContext ...*

```
…
public class MyCustomSecurityContext implements SecurityContext{
   private User user;
   private String scheme;

   public MyCustomSecurityContext(User user, String scheme) {
      this.user = user;
      this.scheme = scheme;
   }
   @Override
   public Principal getUserPrincipal() {
      return this.user;
   }
   @Override
   public boolean isUserInRole(String role) {
      if (user.getRole() != null) {
         return user.getRole().contains(role);
      } return false;
   }
   @Override
   public boolean isSecure() {
      return "https".equals(this.scheme);
   }
   @Override
   public String getAuthenticationScheme() {
      return SecurityContext.BASIC_AUTH;
   }
}
```

```
public class User implements Principal{
   private String firstName, lastName,
                  login, email, password;
   private List<String> role;

   public User(String firstName, String lastName,
               String login, String email,
               String password){
      this.firstName = firstName;
      this.lastName = lastName;
      this.login = login;
      this.email = email;
      this.password = password;
      this.role = new ArrayList<String>();
   };
   …
// getters and setters
   @Override
   public String getName() {
      return this.firstName + " " + this.lastName;
   }
}
```

REST Web Services

AUTHORIZATION *in JAX-RS*

- *add SecurityContext to the RequestContext of SecurityFilter...*

```
@Provider
@Priority(Priorities.AUTHENTICATION)
public class SecurityFilter implements ContainerRequestFilter {
   … // the same variables as it was …
   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException {
      UserService UserService = new UserService();
      User user=null;
      List<String> authHeader = requestContext.getHeaders().get(AUTHORIZATION_HEADER_KEY);
      if (authHeader != null && authHeader.size() > 0) {
         … // the same authorization token decoding as it was. Get username and password …
         if (UserService.userCredentialExists(username, password)) {
            user = UserService.getUser(username);
            String scheme = requestContext.getUriInfo().getRequestUri().getScheme();
            requestContext.setSecurityContext(new MyCustomSecurityContext(user, scheme));
         }
      }
      if ((requestContext.getUriInfo().getPath().contains(SECURED_URL_PREFIX))
      ||(requestContext.getMethod().equals("DELETE"))) {
         if(user!=null) return;
         ErrorMessage errorMessage = new ErrorMessage("User cannot access the resource.", 401,
                                                      "http://myDocs.org");
         Response unauthorizedStatus = Response.status(Response.Status.UNAUTHORIZED)
                                               .entity(errorMessage).build();
         requestContext.abortWith(unauthorizedStatus);
      }
   }
}
```

REST Web Services

AUTHORIZATION *in JAX-RS could be organized via* ***Security Annotations*** *(****@PermitAll****,* ***@DenyAll****,****@RolesAllowed*** *from JavaEE javax.annotation.security package)*

```
…
@Path("/profiles")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@PermitAll
public class ProfileResource {
   private ProfileService profileService = new ProfileService();
   @GET
   public List<Profile> getProfiles(){
      return profileService.getAllProfiles();
   }
   @GET
   @Path("/{profileName}")
   @RolesAllowed("admin")
   public Profile getProfile(@PathParam("profileName") String profileName){
      return profileService.getProfile(profileName);
   }
            …
}
```

In this case, we have to extend our ***SecurityFilter*** *with corresponding role-based access handling…*

REST Web Services

AUTHORIZATION *in JAX-RS*

- *Register **RolesAllowedDynamicFeature** filter*

```
import org.glassfish.jersey.server.ResourceConfig;
import org.glassfish.jersey.server.filter.RolesAllowedDynamicFeature;

@ApplicationPath("webapi")
public class MyAppReg extends ResourceConfig{

   public MyAppReg(){

      register(RolesAllowedDynamicFeature.class);
   }
}
```

- *Annotate resources…*

```
…
@Path("/profiles")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@PermitAll
public class ProfileResource {
            private ProfileService profileService = new ProfileService();
            @RolesAllowed("admin")
            @GET
            @Path("/{profileName}")
            public Profile getProfile(@PathParam("profileName") String profileName){
               return profileService.getProfile(profileName);
            }
            …
}
```

REST Web Services

AUTHORIZATION *in JAX-RS*

```
@Provider
@Priority(Priorities.AUTHENTICATION)
public class SecurityFilter implements ContainerRequestFilter {
   …
   private static final ErrorMessage FORBIDDEN_ErrMESSAGE = new ErrorMessage("Access blocked
                                                    for all users !!!", 403, "http://myDocs.org");
   private static final ErrorMessage UNAUTHORIZED_ErrMESSAGE = new ErrorMessage("User cannot
                                                  access the resource.", 401, "http://myDocs.org");
   @Context private ResourceInfo resourceInfo;
   @Override
   public void filter(ContainerRequestContext requestContext) throws IOException {
      …
      Method resMethod = resourceInfo.getResourceMethod();
      Class<?> resClass = resourceInfo.getResourceClass();

      if(resMethod.isAnnotationPresent(PermitAll.class)){ return; }
      if(resMethod.isAnnotationPresent(DenyAll.class)){
         Response response = Response.status(Response.Status.FORBIDDEN)
                                     .entity(FORBIDDEN_ErrMESSAGE).build();
         requestContext.abortWith(response);
      }
      if(resMethod.isAnnotationPresent(RolesAllowed.class)){
         if(rolesMatched(user,resMethod.getAnnotation(RolesAllowed.class))) return;
         Response response = Response.status(Response.Status.UNAUTHORIZED)
                                     .entity(UNAUTHORIZED_ErrMESSAGE).build();
         requestContext.abortWith(response);
      }
// do the same Annotation check on the Class annotation level with resClass
   }
}
```

REST Web Services

JAX-RS ***Client*** *with* ***Basic Auth*** *.*

```
public class RESTwsClient {
   public static void main(String[] args) {
      Client client = ClientBuilder.newClient();
      WebTarget baseTarget = client.target("http://localhost:8080/RESTws/webapi/");
      WebTarget publicationsTarget = baseTarget.path("publications");
      WebTarget publicationTarget = publicationsTarget.path("{publicationId}");
      WebTarget profilesTarget = baseTarget.path("profiles");
      WebTarget profileTarget = profilesTarget.path("{profileName}");

      String name = "user1";
      String password = "password_user1";
      String authString = name + ":" + password;
      String authStringEnc = Base64.encodeAsString(authString);
System.out.println("Base64 encoded auth string: " + authStringEnc);

      Response getResponse_ = profileTarget.resolveTemplate("profileName", "me")
                .request(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON)
                .header("Authorization", "Basic " + authStringEnc)
                .get();
      Profile profile = getResponse_.readEntity(Profile.class);
System.out.println("ProfileName: "+profile.getProfileName());
      …
   }
}
```

Task 4

Relevant links

This list presents just some relevant links… Obviously, it might be reasonable also to google for some latest tutorial and examples…

Jersey Security
(*https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/security.html*)
JAX-RS Security using Basic and Digest Authentication and Authorization (by web.xml file configuration)
(*https://avaldes.com/jax-rs-security-using-basic-authentication-and-authorization/*),
(*https://avaldes.com/jax-rs-security-using-digest-authentication-and-authorization/*)
Digest Authentication and Authorization
(*https://gist.github.com/lrobb/3745208*)
Jersey (JAX-RS) SecurityContext in action
(*https://simplapi.wordpress.com/2015/09/19/jersey-jax-rs-securitycontext-in-action/*)
Token Based Authentication (*https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication*)
JSON WEB Token
(*https://www.youtube.com/watch?v=_XbXkVdoG_0*),
(*https://www.youtube.com/watch?v=soGRyl9ztjI*),
(*https://www.youtube.com/watch?v=X80nJ5T7YpE*)
JSON Web Token in action with JAX-RS
(*https://abhirockzz.wordpress.com/2016/03/18/json-web-token-in-action-with-jax-rs/*),
(*https://avaldes.com/jax-rs-security-using-json-web-tokens-jwt-for-authentication-and-authorization/*)
OAuth:
(*https://www.youtube.com/watch?v=t4-416mg6iU*),
(*https://www.youtube.com/watch?v=3pZ3Nh8tgTE*)
Securing JAX-RS Services with OAuth 2
(*https://www.youtube.com/watch?v=GYyvfAX7KMY*),
(*https://www.youtube.com/watch?v=ty-2eYCUUVI*)
OAuth 2 sample
(*https://github.com/javaee/jersey/tree/master/examples/oauth2-client-google-webapp*)
Spring Security Hello World Annotation Example
(*http://www.mkyong.com/spring-security/spring-security-hello-world-annotation-example/*)