Best practice for REST token-based authentication with JAX-RS and Jersey

I'm looking for a way to enable authentication token-based in Jersey .

I was trying not to use the particular framework. Is that possible?

I would:

A user at the time to sign up for my web service, my web service generates a token, send it to the client, and the client will retain. Then the client for each request will not send user or password, but this token.

I was thinking of using custom filter for each request and the `@PreAuthorize("hasRole('ROLE')")` but I just thought that this causes a lot of requests to the database to see if the token is right.

Or not create filter and in each request put a param token?
So each API first check token and after execute something to retrieve resource.

java rest jersey-2.0

edited Nov 6 '14 at 12:51
Cássio Mazzochi Molin
999 1 11 29

asked Nov 6 '14 at 10:26
Antonio Balduzzi
936 1 4 17

Do you really need a token-based authentication? If not, consider using and basic authentication and always use HTTPS. – Cássio Mazzochi Molin Nov 6 '14 at 10:39

Yes because i would not send user and password everytime, and store user pass in the client device. Because the client is a device – Antonio Balduzzi Nov 6 '14 at 10:41

mobile device.. – Antonio Balduzzi Nov 6 '14 at 10:50

1 Answer

How token-based authentication works

A token is a piece of data generated by the server which identifies a user.

At a first look, token-based authentication follow these steps:

1. The client sends its credentials (username and password) to the server.
2. The server authenticates them and generates a fixed length token with an expiration date.
3. The server stores the previously generated token in some storage with user identifier, such as a database or a map in memory.
4. The server sends previously generated token to the client.
5. In every request, the client sends that token to the server.
6. The server, in each request, extracts the token from incoming request, looks up the user identifier with the token to obtain the user information to do the authorization.
7. If the token is expired, the server generates another token and send back to the client.

What you can do with JAX-RS 2.0 (Jersey and RESTEasy)

If you wouldn't like to reinvent the wheel, try PicketLink which provides out of the box security solutions for Java applications. Otherwise, you can write your own solution as described below.

It's important mention that if you are using a token-based authentication, you are not relying on the standard Java EE Web application security mechanisms offered by the Servlet container and configurable via application's `web.xml` descriptor.

Authenticate a user with their username and password and issue a token

Create a REST endpoint which receives and validates the credentials (username and password) and issue a token for the user:

```
@Path("/authentication")
public class AuthenticationEndpoint {

    @POST
    @Produces("application/json")
    @Consumes("application/x-www-form-urlencoded")
    public Response authenticate(@FormParam("username") String username,
@FormParam("password") String password) {

        try {

            // Authenticate the user using the credentials provided
            authenticate(username, password);

            // Issue a token for the user
            String token = issueToken(username);

            // Return the token on the response
            return Response.ok(token).build();

        } catch (Exception e) {
            return Response.status(Response.Status.UNAUTHORIZED).build();
        }
    }

    private void authenticate(String username, String password) throws Exception {
        // It's up to you (authenticate against a database, LDAP or whatever)
        // Throw an Exception if the credentials are invalid
    }

    private String issueToken(String username) {
        // It's up to you (use a random String persisted to the database or a JWT token)
        // The issued token must be associated to a user
    }
}
```

If any exceptions happen when validating the credentials, a response with status `401 UNAUTHORIZED` will be returned.

If the credentials are successfully validated, a response with status `200 OK` will be returned and the issued token is sent to the client on the response. The client must send that token to the server in every request.

Instead of form params, you can wrap the username and the password into a class:

```
public class Credentials implements Serializable {

    private String username;
    private String password;

    // Getters and setters omitted
}
```

And consume it as JSON:

```
    @POST
    @Produces("application/json")
    @Consumes("application/json")
    public Response authenticate(Credentials credentials) {

        String username = credentials.getUsername();
        String password = = credentials.getPassword();

        // Authenticate the user, issue a token and return a response
    }
```

Extract the token from the request and validate it

The client should send the token on the standard HTTP `Authorization` header of the request. For example:

```
Authorization: Bearer <token-goes-here>
```

Note that the name of the standard HTTP header is unfortunate because it carries *authentication* information, not *authorization*.

JAX-RS provides `@NameBinding`, a meta-annotation used to create name-binding annotations for filters and interceptors:

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(value = RetentionPolicy.RUNTIME)
@NameBinding
public @interface Secured { }
```

The defined name-binding annotation `@Secured` will be used to decorate a filter class, which implements `ContainerRequestFilter`, allowing you to handle the request. The `ContainerRequestContext` helps you to get information about the request:

```
@Secured
@Provider
@Priority(Priorities.AUTHENTICATION)
public class AuthenticationFilter implements ContainerRequestFilter {
    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {

        // Get the HTTP Authorization header from the request
        String authorizationHeader = 
requestContext.getHeaderString(HttpHeaders.AUTHORIZATION);

        // Check if the HTTP Authorization header is present and formatted correctly 
        if (authorizationHeader == null || !authorizationHeader.startsWith("Bearer ")) {
            throw new NotAuthorizedException("Authorization header must be provided");
        }

        // Extract the token from the HTTP Authorization header
        String token = authorizationHeader.substring("Bearer".length()).trim();

        try {

            // Validate the token by checking if it was issued by the server and if it's
not expired

        } catch (Exception e) {

requestContext.abortWith(Response.status(Response.Status.UNAUTHORIZED).build());
        }
    }
}
```

If any problems happen during the token validation, a response with status `401 UNAUTHORIZED` will be returned.

Otherwise, the request will proceed to an endpoint.

Securing your REST endpoints

Bind the filter to your endpoints methods or classes by annotating them with the `@Secured` annotation created above. For the methods and/or classes which are annotated, the filter will be executed. It means that these endpoints only will be reached if the request is performed with a valid token.

If some methods or classes do not need authentication, simply do not annotate them.

```
@Path("/")
public class MyEndpoint {

    @GET
    @Path("{id}")
    @Produces("application/json")
    public Response myUnsecuredMethod(@PathParam("id") Long id) {
        ...
    }

    @DELETE
    @Secured
    @Path("{id}")
    @Produces("application/json")
    public Response mySecuredMethod(@PathParam("id") Long id) {
        ...
    }
}
```

In the example above, the filter will be executed only for `mySecuredMethod(Long)` because it's annotated with `@Secured`.

Identifying the current user

It's very likely you will need to know the user who is performing the request within your REST endpoints. The easiest way to achieve it is setting a `SecurityContext` on the `ContainerRequestContext` within your `ContainerRequestFilter.filter(ContainerRequestContext)` method.

Implement the `SecurityContext.getUserPrincipal()`, returning a `Principal` instance. The `Principal` 's name is the username of the user you issued the token for. You will have to know it when validating the token.

```
    requestContext.setSecurityContext(new SecurityContext() {

        @Override
        public Principal getUserPrincipal() {

            return new Principal() {

                @Override
                public String getName() {
                    return username;
                }
            };
        }

        @Override
        public boolean isUserInRole(String role) {
            return true;
        }
        @Override
        public boolean isSecure() {
            return false;
        }

        @Override
        public String getAuthenticationScheme() {
            return null;
        }
    });
```

Inject a proxy of the `SecurityContext` in any REST endpoint class:

```
    @Context
    SecurityContext securityContext;
```

The same can be done in a method:

```
    @GET
    @Secured
    @Path("{id}")
    @Produces("application/json")
    public Response myMethod(@PathParam("id") Long id, @Context SecurityContext
 securityContext) {
        ...
    }
```

And get the `Principal` :

```
    Principal principal = securityContext.getUserPrincipal();
    String username = principal.getName();
```

How to issue a token

Random string

A token can be issued by generating a random string and persisting it to a database with an expiration date and with a user identifier associated to it. A good example of how to generate a random string in Java can be seen here:

```
    Random random = new SecureRandom();
    String token = new BigInteger(130, random).toString(32);
```

JSON Web Tokens (JWT)

Alternatively, you can use JSON Web Tokens (JWT) which is a self-contained token: can store the user identifier, an expiration date and whatever you want (do not store passwords on it).

There are a few Java libraries to issue and validate JWT tokens (have a look here and here).

As JWT uses signatures, the integrity of the tokens can be easily checked. And you won't need to persist the tokens if you don't need to track them.

Although, persisting the token will give you the possibility of invalidate and revoke the access of a token. But always consider removing the old tokens in order to prevent the database from growing indefinitely.

Additional information

1. It doesn't matter which type of authentication you are using. **Always use HTTPS** and prevents the man-in-the-middle attack.
2. Take a look at this question from Information Security for more information about tokens.
3. Have a look here for more details about JSON Web Tokens (JWT).
4. Here you will find some useful information about token-based authentication.
5. Apache DeltaSpike provides portable CDI extensions such as a security module, which can be used to secure REST applications.
6. Interested in an OAuth 2.0 protocol implementation in Java? Check the Apache Oltu project.

edited 2 days ago answered Nov 6 '14 at 11:17
Cássio Mazzochi Molin
999 1 11 29

if i use NameBinding i need to create a custom filter right? and another question, in the class LoggingFilter in you example, i put all rule for the token right? – Antonio Balduzzi Nov 6 '14 at 11:27

1 Yes. Create a class implementing `ContainerRequestFilter` and implement the method `filter(ContainerRequestContext)` . From the `ContainerRequestContext` , you can get information about the request. – Cássio Mazzochi Molin Nov 6 '14 at 11:42

and in ContainerRequestContext i put all the code for generating token or update token and check token? – Antonio Balduzzi Nov 6 '14 at 13:11

1 Use `ContainerRequestContext` to get the token from the request and validate it. Use `ContainerResponseContext` to put the token on the response or update it. – Cássio Mazzochi Molin Nov

6 '14 at 13:16

thanks! and in the web.xml i need to add something? – Antonio Balduzzi Nov 6 '14 at 13:18