How token-based authentication works

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 follow these steps:

1. The client sends their credentials (username and password) to the server.
2. The server authenticates the credentials and, if they are valid, generate 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: The step 3 is not required if the server has issued a signed token (such as JWT, which allows you to perform *stateless* authentication).

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

This solution uses only the JAX-RS 2.0 API, *avoiding any vendor specific solution*. So, it should work with JAX-RS 2.0 implementations, such as Jersey, RESTEasy and Apache CXF.

It is worthwhile to mention that if you are using 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. It's a custom authentication.

Authenticating a user with their username and password and issuing a token

Create a JAX-RS resource method which receives and validates the credentials (username and password) and issue a token for the user:

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

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public Response authenticateUser(@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.FORBIDDEN).build();
        }
    }

    private void authenticate(String username, String password) throws
Exception {
        // Authenticate against a database, LDAP, file or whatever
        // Throw an Exception if the credentials are invalid
    }

    private String issueToken(String username) {
        // Issue a token (can be a random String persisted to a database or a
JWT token)
        // The issued token must be associated to a user
        // Return the issued token
    }
}
```

If any exceptions are thrown when validating the credentials, a response with the status `403` (Forbidden) will be returned.

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

When consuming `application/x-www-form-urlencoded`, the client must to send the credentials in the following format in the request payload:

```
username=admin&password=123456
```

Instead of form params, it's possible to 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 then consume it as JSON:

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response authenticateUser(Credentials credentials) {

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

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

Using this approach, the client must to send the credentials in the following format in the payload of the request:

```
{
  "username": "admin",
  "password": "123456"
}
```

Extracting the token from the request and validating it

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

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

The name of the standard HTTP header is unfortunate because it carries *authentication* information, not *authorization*. However, it's the standard HTTP header for sending credentials to the server.

JAX-RS provides `@NameBinding`, a meta-annotation used to create other annotations to bind filters and interceptors to resource classes and methods. Define a `@Secured` annotation as following:

```
@NameBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface Secured { }
```

The above defined name-binding annotation will be used to decorate a filter class, which implements `ContainerRequestFilter`, allowing you to intercept the request before it be handled by a resource method. The `ContainerRequestContext` can be used to access the HTTP request headers and then extract the token:

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

    private static final String AUTHENTICATION_SCHEME = "Bearer";

    @Override
    public void filter(ContainerRequestContext requestContext) throws
IOException {

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

        // Validate the Authorization header
        if (!isTokenBasedAuthentication(authorizationHeader)) {
            abortWithUnauthorized(requestContext);
            return;
        }

        // Extract the token from the Authorization header
        String token = authorizationHeader

.substring(AUTHENTICATION_SCHEME.length()).trim();

        try {

            // Validate the token
            validateToken(token);

        } catch (Exception e) {
            abortWithUnauthorized(requestContext);
        }
    }

    private boolean isTokenBasedAuthentication(String authorizationHeader) {

        // Check if the Authorization header is valid
        // It must not be null and must be prefixed with "Bearer" plus a
whitespace
        // Authentication scheme comparison must be case-insensitive
        return authorizationHeader != null &&
authorizationHeader.toLowerCase()
                    .startsWith(AUTHENTICATION_SCHEME.toLowerCase() + " ");
    }

    private void abortWithUnauthorized(ContainerRequestContext
requestContext) {

        // Abort the filter chain with a 401 status code
        // The "WWW-Authenticate" is sent along with the response
        requestContext.abortWith(
                Response.status(Response.Status.UNAUTHORIZED)
                        .header(HttpHeaders.WWW_AUTHENTICATE,
AUTHENTICATION_SCHEME)
                        .build());
    }

    private void validateToken(String token) throws Exception {
        // Check if it was issued by the server and if it's not expired
        // Throw an Exception if the token is invalid
    }
}
```

If any problems happen during the token validation, a response with the status `401` (Unauthorized) will be returned. Otherwise the request will proceed to a resource method.

Securing your REST endpoints

To bind the authentication filter to resource methods or resource classes, annotate them with the `@Secured` annotation created above. For the methods and/or classes that are annotated, the filter will be executed. It means that such endpoints will *only* 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("/example")
public class ExampleResource {

    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response myUnsecuredMethod(@PathParam("id") Long id) {
        // This method is not annotated with @Secured
        // The authentication filter won't be executed before invoking this
method
        ...
    }

    @DELETE
    @Secured
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response mySecuredMethod(@PathParam("id") Long id) {
        // This method is annotated with @Secured
        // The authentication filter will be executed before invoking this
method
        // The HTTP request must be performed with a valid token
        ...
    }
}
```

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

Identifying the current user

It's very likely that you will need to know the user who is performing the request agains your REST API. The following approaches can be used to achieve it:

Overriding the security context of the current request

Within your `ContainerRequestFilter.filter(ContainerRequestContext)` method, a new `SecurityContext` instance can be set for the current request. Then override the `SecurityContext.getUserPrincipal()`, returning a `Principal` instance:

```
final SecurityContext currentSecurityContext =
requestContext.getSecurityContext();
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 currentSecurityContext.isSecure();
    }

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

Use the token to look up the user identifier (username), which will be the `Principal`'s name.

Inject the `SecurityContext` in any JAX-RS resource class:

```
@Context
SecurityContext securityContext;
```

The same can be done in a JAX-RS resource method:

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

And then get the `Principal`:

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

Using CDI (Context and Dependency Injection)

If, for some reason, you don't want to override the `SecurityContext`, you can use CDI (Context and Dependency Injection), which provides useful features such as events and producers.

Create a CDI qualifier:

```
@Qualifier
@Retention(RUNTIME)
@Target({ METHOD, FIELD, PARAMETER })
public @interface AuthenticatedUser { }
```

In your `AuthenticationFilter` created above, inject an `Event` annotated with `@AuthenticatedUser`:

```
@Inject
@AuthenticatedUser
Event<String> userAuthenticatedEvent;
```

If the authentication succeeds, fire the event passing the username as parameter (remember, the token is issued for a user and the token will be used to look up the user identifier):

```
userAuthenticatedEvent.fire(username);
```

It's very likely that there's a class that represents a user in your application. Let's call this class `User`.

Create a CDI bean to handle the authentication event, find a `User` instance with the correspondent username and assign it to the `authenticatedUser` producer field:

```
@RequestScoped
public class AuthenticatedUserProducer {

    @Produces
    @RequestScoped
    @AuthenticatedUser
    private User authenticatedUser;

    public void handleAuthenticationEvent(@Observes @AuthenticatedUser String 
username) {
        this.authenticatedUser = findUser(username);
    }

    private User findUser(String username) {
        // Hit the the database or a service to find a user by its username 
and return it
        // Return the User instance
    }
}
```

The `authenticatedUser` field produces a `User` instance that can be injected into container managed beans, such as JAX-RS services, CDI beans, servlets and EJBs. Use the following piece of code to inject a `User` instance (in fact, it's a CDI proxy):

```
@Inject
@AuthenticatedUser
User authenticatedUser;
```

Note that the CDI `@Produces` annotation is *different* from the JAX-RS `@Produces` annotation:

- CDI: `javax.enterprise.inject.Produces`
- JAX-RS: `javax.ws.rs.Produces`

Be sure you use the CDI `@Produces` annotation in your `AuthenticatedUserProducer` bean.

The key here is the bean annotated with `@RequestScoped`, allowing you to share data between filters and your beans. If you don't wan't to use events, you can modify the filter to store the authenticated user in a request scoped bean and then read it from your JAX-RS resource classes.

Compared to the approach that overrides the `SecurityContext`, the CDI approach allows you to get the authenticated user from beans other than JAX-RS resources and providers.

Supporting role-based authorization

Please refer to my other answer for details on how to support role-based authorization.

Issuing tokens

A token can be:

- **Opaque:** Reveals no details other than the value itself (like a random string)
- **Self-contained:** Contains details about the token itself (like JWT).

See details below:

Random string as token

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

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

JWT (JSON Web Token)

JWT (JSON Web Token) is a standard method for representing claims securely between two parties and is defined by the RFC 7519.

It's a self-contained token and it enables you to store details in *claims*. These claims are stored in the token payload which is a JSON encoded as Base64. Here are some claims registered in the RFC 7519 and what they mean (read the full RFC for further details):

- `iss`: Principal that issued the token.
- `sub`: Principal that is the subject of the JWT.
- `exp`: Expiration date for the token.
- `nbf`: Time on which the token will start to be accepted for processing.
- `iat`: Time on which the token was issued.
- `jti`: Unique identifier for the token.

Be aware that you must not store sensitive data, such as passwords, in the token.

The payload can be read by the client and the integrity of the token can be easily checked by verifying its signature on the server. The signature is what prevents the token from being tampered with.

You won't need to persist JWT tokens if you don't need to track them. Althought, by persisting the tokens, you will have the possibility of invalidating and revoking the access of them. To keep the track of JWT tokens, instead of persisting the whole token on the server, you could persist the token identifier (`jti` claim) along with some other details such as the user you issued the token for, the expiration date, etc.

When persisting tokens, always consider removing the old ones in order to prevent your database from growing indefinitely.

Using JWT

There are a few Java libraries to issue and validate JWT tokens such as:

- jjwt
- java-jwt
- jose4j

To find some other great resources to work with JWT, have a look at http://jwt.io.

Handling token refreshment with JWT

Accept *only* valid (and non-expired) tokens for refreshment. It's responsability of the client to refresh the tokens before the expiration date indicated in the `exp` claim.

You should prevent the tokens from being refreshed indefinitely. See below a few approaches that you could consider.

You could keep the track of token refreshment by adding two claims to your token (the claim names are up to you):

- `refreshLimit`: Indicates how many times the token can be refreshed.
- `refreshCount`: Indicates how many times the token has been refreshed.

So only refresh the token if the following conditions are true:

- The token is not expired (`exp >= now`).
- The number of times that the token has been refreshed is less than the number of times that the token can be refreshed (`refreshCount < refreshLimit`).

And when refreshing the token:

- Update the expiration date (`exp = now + some-amount-of-time`).
- Increment the number of times that the token has been refreshed (`refreshCount++`).

Alternatively to keeping the track of the number of refreshments, you could have a claim that indicates the *absolute expiration date* (which works pretty similar to the `refreshLimit` claim described above). Before the *absolute expiration date*, any number of refreshments is acceptable.

Another approach involves issuing a separate long-lived refresh token that is used to issue short-lived JWT tokens.

The best approach depends on your requirements.

Handling token revocation with JWT

If you want to revoke tokens, you must keep the track of them. You don't need to store the whole token on server side, store only the token identifier (that must be unique) and some metadata if you need. For the token identifier you could use UUID.

The `jti` claim should be used to store the token identifier on the token. When validating the token, ensure that it has not been revoked by checking the value of the `jti` claim against the token identifiers you have on server side.

For security purposes, revoke all the tokens for a user when they change their password.

Additional information

- It doesn't matter which type of authentication you decide to use. **Always** do it on the top of a HTTPS connection to prevent the man-in-the-middle attack.
- Take a look at this question from Information Security for more information about tokens.
- In this article you will find some useful information about token-based authentication.