REST Authentication

For DSpace 7 the REST authentication has been rewritten to use JSON Web tokens instead of Tomcat session ID's. This was done to support stateless sessions. The implementation still uses Spring Security similar to DSpace 6.

This documentation is a historical record for how the REST API authentication was designed and why JSON Web Tokens are used.

While this page may be useful to understand the underlying design, the latest documentation for **how to authenticate** via the REST API is found in our REST Contract: https://github.com/DSpace/RestContract/blob/main/authentication.md The authentication examples found below may be outdated

Authenticate

To authenticate yourself, you have to send a POST request to the `/api/authn/login` endpoint with the following parameters:

parameter	value
user	email/id of user
password	password of user

Example call with curl:

```
# Login: test@dspace.com , Password: p4ssword
curl -v -X POST --data "user=test%40dspace.com&password=p4ssword" "https://{dspace.server.url}/api/authn/login"
-H "X-XSRF-TOKEN: {csrf-token}"
```

NOTE: a login request first requires obtaining a valid CSRF token from the REST API. See https://github.com/DSpace/RestContract/blob/main/csrf-tokens.md

This call will return a JWT (JSON Web Token) in the response in the `Authorization` header according to the bearer scheme. This token has to be used in subsequent calls to provide your authentication details. For example:

```
curl -v "https://{dspace.server.url}/api/core/items" -H "Authorization: Bearer eyJhbG...COdbo"
```

See also https://github.com/DSpace/RestContract/blob/main/authentication.md

Login using the HAL-browser

The HAL browser has been extended to provide a login form and to authenticate subsequent requests. You can find the login form in the top navigation menu where you can enter your credentials. After supplying valid credentials the token will be stored in a cookie and every request will get this token from the cookie and send it in the authorization header.

Pass the token

For DSpace to detect the token it has to be send in the `Authorization` header with the Bearer schema, that is prepend the token with "Bearer" then leave a space and paste the token.

Keep in mind, all modifying requests (POST, PUT, PATCH, DELETE) also require sending a separate CSRF Token in the `X-XSRF-TOKEN` header. See https://github.com/DSpace/RestContract/blob/main/csrf-tokens.md

Authentication Status

The authentication status can be checked by sending your received token to the status endpoint in the `Authorization` header:

```
curl -v "https://{dspace.server.url}/api/authn/status" -H "Authorization: Bearer eyJhbG...COdbo"
```

This will return the authentication status, E.G.:

```
{
  "okay" : true,
  "authenticated" : true,
  "type" : "status",
  "_links" : {
    "eperson" : {
      "href" : "http://localhost:8080/dspace7-rest/api/eperson/epersons/2245f2c5-1bed-414b-a313-3fd2d2ec89d6"
    }
  },
  "_embedded" : {
    "eperson" : {
      "uuid" : "2245f2c5-1bed-414b-a313-3fd2d2ec89d6",
      "email" : "test@dspace.com",
      ...
      }
    }
  }
}
```

Fields

Field	Meaning
Okay	True if REST API is up and running, should never return false
Authenticated	True if the token is valid, false if there was no token or the token wasn't valid
Type	Type of the endpoint, "status" in this case
_links	returns a link to the authenticated eperson
_embedded	Embeds the authenticated eperson

Logout

To logout and invalidate the token, send the token in the Authorization header with the bearer scheme to the following endpoint:

/api/authn/logout

E.G.

```
# NOTE: Logout must be done via POST
curl -v -X POST "https://{dspace.server.url}/api/authn/logout" -H "Authorization: Bearer eyJhbG...COdbo" -H "X-XSRF-TOKEN: {csrf-token}"
```

This will log the user out on every device or browser.

See also https://github.com/DSpace/RestContract/blob/main/authentication.md

JSON Web Token

The authentication token is a JSON Web Token (JWT) and is base64url encoded. For more information about JWT see this page: https://jwt.io/introduction/

By default the JWT token will have a couple of claims already, which we can see if we decode the token:

Claim	Data
eid	Contains the id of the eperson
sg	Contains the id's of the special groups to which a user belongs
exp	Contains the expiration date when a token will expire

Add extra claims

To add a custom additional claim, you should implement a Spring bean which implement the JWTClaimProvider interface. Spring will scan for beans implementing that interface and use them to automatically add new claims to the tokens.

The JWTClaimProvider interface requires three methods to be implemented:

`getKey(): String`

This method should return a string, this string will be used as key for the claim (for example "eid" for the eperson id claim)

`getValue(Context, HttpServletRequest): Object`

This method should return the value of the claim, This can be any object, as long as it is Serialisable.

`parseClaim(Context, HttpServletRequest, JWTClaimSet)`

This method should parse the claim when someone presents a token. In this method you should handle what has to happen with it (for example setting special groups on the context object)

NOTE: add `@Component` to your ClaimProviders so Spring can find them.

Refresh Token

Tokens are only valid for a configurable amount of time (see below). When a token is about to expire (the timestamp provided in the exp claim), you can request a new token with a new expiration time (by default 30 minutes). To do so send the token to the login endpoint without "user" and "password" parameters. As a response you'll get a new freshly issued token (again in the Authorization header of the response).

E.G.

```
curl -v "http://{spring-rest.url}/api/authn/login" -H "Authorization: Bearer eyJhbG...COdbo"
```

Which will return something like this:

```
HTTP/1.1 200 OK
Authorization: Bearer edoDfG...S0df
```

Now you can use this new token to continue making authenticated requests.

Configuration properties

The new stateless authentication introduced a few new properties that can be configured, these can all be found under `modules/authentication.cfg`:

jwt.token.secret	Manually define a key that will be used (in combination with other strings) to sign the tokens. If this property is empty, a random key will be generated. Note that if you want to run DSpace in a cluster with multiple instances this has to be configured and every instance has to use the same key. It is also possible to pass this property with a value as an environment variable.
jwt.encryption.enabled	Boolean property, defaults to false. If enabled the tokens will be encrypted and unreadable client-side. As a downside enabling this makes the tokens a bit larger which will make the size of requests a bit larger, another disadvantage is not being able to use the data that is inside the token. This means for example that the client cannot read the expiration claim and has to guess when it should refresh its token.
jwt.encryption.secret	Key to use if encryption for JWT is enabled. If none is specified and encryption is enabled, DSpace will generate a random one. In a clustered setup, the encryption key should be the same on all instances.
jwt.token.expiration	Enter the period in minutes that a token should be valid, by default this is 30

Running DSpace in a clustered setup

One of the biggest advantages of this stateless authentication is that we can scale DSpace horizontally by running multiple instances of DSpace side-by-side without complex configuration. Each node in the cluster will be able to understand the tokens. To do so a few properties have to be configured:

- jwt.token.secret: This property has to be the same on every instance of DSpace. It's best to choose a long, non trivial secret for extra security. Remember that properties can also be set through environment variables.
- if jwt.encryption.enabled is set to true then jwt.encryption.secret also has to be configured and the same key has to be used on every instance.

Construction of signing key

The signing key used to sign and validate tokens is unique per eperson session. The signing key is constructed by

- a random generated salt per user +
- the server jwt.token.secret or a random one if empty +

The session salt is saved in the EPerson table in the database and is used for:

- Invalidating tokens: When someone logs out the salt will be removed from the database, so that tokens won't be valid anymore since the sign key can't be constructed anymore.
- Making sure the signing key has a valid length: The salt is always 32 bytes and the key to sign is required to be >= 32 bytes

As long as a user refreshes his tokens before they expire, the session salt will not change. Once all tokens are expired and have not been refreshed (or if the user called the logout endpoint), the session salt will change on the next login.