bam-server Documentation

Release v1.0

Julien Delafontaine @ CHUV

Jun 12, 2017

Contents

1 Contents **1**

1.1 What is bam-server . . . 1
1.2 How it works . . . 1
1.3 Installation guidelines . . . 3
1.3.1 Requirements . . . 3
1.3.2 Running in dev mode . . . 4
1.3.3 Deployment . . . 4
1.3.4 Configuration . . . 4
1.3.5 Building from source . . . 5
1.4 BAM query API . . . 5
1.4.1 Authorization . . . 5
1.4.2 API . . . 6
1.5 Authorization protocol . . . 7
1.5.1 HMAC protocol . . . 7
1.5.2 RSA protocol . . . 8
1.5.3 JWT example . . . 8
1.5.4 Supported algorithms . . . 9
1.6 Local database . . . 9
1.6.1 Schema . . . 9
1.6.2 Fields description . . . 10
1.6.3 Some test data . . . 10
1.7 Management API . . . 11
1.7.1 Add/remove an app . . . 11
1.7.2 Add/remove users . . . 12
1.7.3 Add/remove samples . . . 12
1.7.4 Add/remove an attribution . . . 13
1.8 Implementation examples . . . 14
1.8.1 Using IGV.js as reads viewer . . . 14
1.8.2 Example of using Auth0 as authorization server . . . 14

2 Indices and tables **17**

CHAPTER 1

Contents

What is bam-server

Bam-server is a microservice with a REST API that allows to query regions of BAM files remotely and securely, according to user permissions. In particular, it

- Never moves or copies the original file, but sends only the requested bits through HTTP;
- Is compatible with any authorization server that returns JSON Web Tokens (*does not work with cookies*).
- Integrates very well with IGV.js.

A typical application is the display of local read alignments for variant calling quality control, where the subject is human and thus the data must be kept private.

An example of client web application that uses IGV.js to display read alignments from protected BAM files can be found here: bam-server-client-react.

Bam-server is written in Scala using the Play framework.

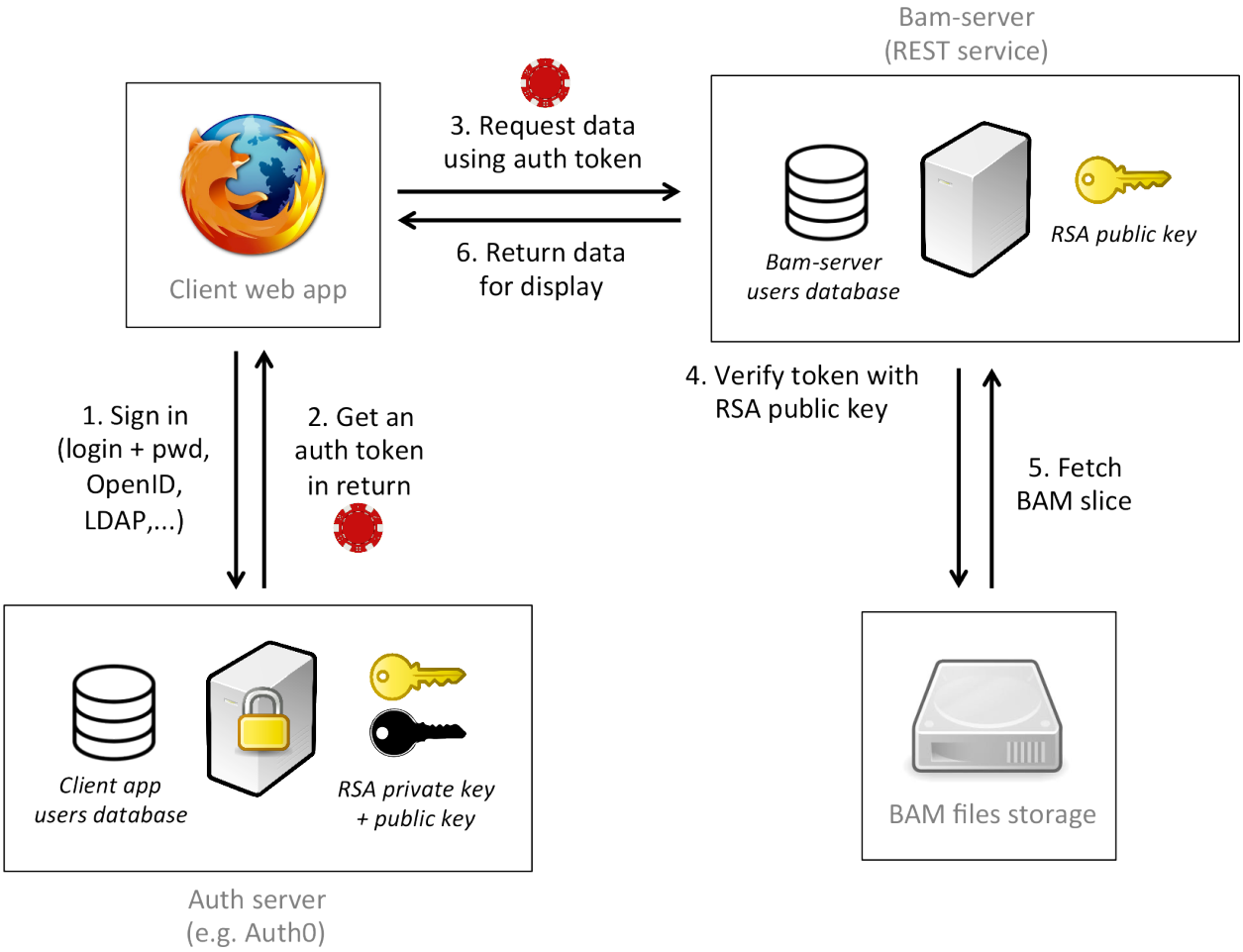
How it works

Bam-server in itself represents only the resource server in the authorization workflow (see figure below):

1. The client app connects to its authorization server to request an authorization token;
2. The auth server returns a JSON Web Token (JWT) signed with either RSA or HMAC. The JWT must contain at least an identifier for the client app (typically the *iss* claim), and an identifier for the user that sends the request (*name* by default, but configurable).
3. The client app requests a portion of a BAM file for a given sample from the bam-server using the REST API.
4. Bam-server verifies the token using either the RSA public key or the HMAC shared secret, then checks the user and app identifiers against its own users database.
5. Bam-server extracts the requested slice from the BAM file corresponding to the sample and

Fig. 1.1: Demo interface using bam-server to display read alignments.

6. returns it to the client app.



Notes:

- It is up to the auth server to define valid users. The local database of bam-server only maps user identifiers as returned by the auth server, to sample identifiers and the corresponding BAM files.
- Client apps can access bam-server remotely, but for the moment, bam-server itself can only access files that are located on the same machine as itself (the location of BAM files is defined using a configuration variable *BAM_PATH*, see *Installation guidelines*). One cannot provide a URL as a file path in the database.

Installation guidelines

Bam-server was written in Scala using the Play framework (2.5.4). Here we summarize the standard deployment guidelines for a Play application. For more details, refer to the Play docs.

Requirements

- Java RE 8+

To work from source:

- Scala 2.11.7+

- sbt 0.13.11+

Running in dev mode

Running the development server can be useful either to quickly test the application before setting up a production installation, or when you want to edit the source code. From source, in the root directory:

```
sbt run
```

By default it gets served at *localhost:9000* (try in your browser's address bar).

To use the test configuration instead of the default (recommended):

```
sbt run -Dconfig.resource=application.test.conf
```

To run the tests:

```
sbt test
```

Tests use a built-in H2 database with hard-coded data that gets destroyed and recreated on each run. Its contents can be found in */conf/evolutions/default/1.sql*.

Deployment

Get the latest release from GitHub. A precompiled release for v1.0 is available here, or you can *build from source*. Move the build archive to destination, decompress, enter the directory, then launch the server (netty):

```
./bin/bam-server
```

By default it gets served at *localhost:9000* (try in your browser's address bar). When it starts, a file *RUNNING_PID* is created in the root directory that contains the ID of the running process. To stop the application from running, kill this process and do not forget to remove the file:

```
cat RUNNING_PID | xargs kill -9
rm RUNNING_PID
```

You can configure a proxy to make it available from the outside world. For example with Apache:

```
<VirtualHost *:443>
    ...
    ProxyPass         /bamserver  http://localhost:9000
    ProxyPassReverse  /bamserver  http://localhost:9000
    ...
</Virtualhost>
```

The service becomes available at `https://<host>/bamserver/`. Apache can take care itself of protecting external transactions with HTTPS. To use another proxy than Apache, refer to Play HTTPServer docs.

Configuration

When deploying, you will certainly want to configure at least the server port, the application secret, or completely replace the configuration file (recommended), for instance:

```
./bin/bam-server -v \
    -Dconfig.file=../application.prod.conf \
    -Dhttp.port=8870 \
    -Dhttps.port=8871
```

Settings can be edited in */conf/application.conf*, but we recommend specifying a new config file with `-Dconfig.file=<path/to/config>` as shown above.

All you want to know about configuration files and options for Play is described there.

For bam-server in particular,

- You must indicate `env.BAM_PATH` to tell where it can find BAM files for reading.
- It is set to use SQLite. You can change the database settings in the `db` section.
- You will probably want to configure CORS in `play.filters.cors` to white-list your client and restrict others.

A default SQLite database is provided with the project source (*/resources/db/bam-server*), already with the schema and a few test data.

Building from source

```
git clone --depth 1 https://github.com/jdelafon/bam-server-scala.git
cd bam-server-scala/
sbt dist
```

It creates a distribution archive under */target/universal/*.

BAM query API

Bam-server can return portions of BAM files using 3 different strategies:

1. *Extracting a range of bytes* (IGV.js uses that);
2. *Using samtools* (if available);
3. *Using Picard-tools/htsjdk* to return reads in JSON format.

Authorization

Only registered users (present in bam-server's database in the *users* table) can use these routes.

Users are identified using an Authorization header to be added to the request, containing a signed Bearer token (JWT, see *Authorization protocol*). All requests expect such a header. For example:

```
curl -i -H "Authorization: Bearer xxxx.yyyy.zzzz" -X POST -d '{"sample": "SAMPLE1"}␣
↪http://localhost:9000/bai
curl -i -H "Authorization: Bearer xxxx.yyyy.zzzz" http://localhost:9000/bai/SAMPLE1
```

Alternatively, all request can accept a URL argument *?token=<JWT>* to pass the token. This mode is not recommended but sometimes necessary, depending on client library constraints. For example:

```
curl -i http://localhost:9000/bai/SAMPLE1?token=xxxx.yyyy.zzzz
```

API

Let *:sample/<sample>* be a sample identifier in the database;

Let *<range>* be a bytes range, formatted as "123-456";

Let *<region>* be a genomic region, formatted as "chr1:10000-20000". The chromosome names are those referenced in the BAM file header.

```
GET /
```

Says "BAM server operational.", just to test if the server is listening.

BAM index

```
POST /bai                                   # data = '{"sample": "<sample>"}'
GET /bai/:sample
```

Returns the content of the index (.bai) for that BAM file.

Range query

```
POST /bam/range                             # data = '{"sample": "<sample>"}'
GET /bam/range/:sample
```

Returns the content of the BAM file, expecting an HTTP Range header to extract only the bytes range of interest - likely based on the BAM index. The bytes range can also be passed as an argument to the request (*?range=<range>*). Return the content as binary.

Using samtools

```
POST /bam/samtools?region=<region>          # data = '{"sample": "<sample>"}'
GET /bam/samtools/:sample?region=<region>
```

Uses samtools (if available) to extract the region (`samtools view -hb <bam> <region>`). Return the content as binary.

Reads in JSON format

```
POST /bam/json?region=<region>              # data = '{"sample": "<sample>"}'
GET /bam/json/:sample?region=<region>
```

Returns the reads for the given region in JSON format, using the htsjdk library. The fields correspond to the SAM file columns:

```
[
  {
     "name": "HISEQ:206:C8E95ANXX:3:2113:2451:6639",    // read name
     "flag": 99,
     "chrom": "chr1",        // reference name
     "start": 1234,          // leftmost mapping position
     "end": 1334,            // rightmost mapping position
```

```
    "mapq": 50,             // mapping quality
    "cigar": "101M",        // cigar string
    "rnext": "=",           //
    "pnext": 4567,          //
    "tlen": 283,            // template length, aka insert size
    "seq": "AATTAGGA...",   // [ACGTN=.],
    "qual": "AB<B@G>F..."   // per-base quality
  },
...
]
```

Authorization protocol

To ensure that only registered users have access only to the data that belongs to them, all requests must include a Json Web Token (JWT) in their "Authorization" header. For example:

```
curl -i -H "Authorization: Bearer xxxx.yyyy.zzzz" http://localhost:9000/bai/SAMPLE1
```

The token payload must include at least (see *JWT_example*):

- The user identifer under claim *"name"* (configurable);
- The app identifier under claim *"iss"*.

The authorization process works as follows:

1. The user ("name") is matched against the *users* table in the database to make sure he is registered.
2. The app name ("iss") is matched against the *apps* table in the database, so as to retreive the corresponding verification key from the *key* column and the *algorithm* used to sign the token.
3. The JWT is verified using the key.

Bam-server can work with two kinds of signature algorithms:

- **HMAC** (shared secret)
- **RSA** (asymmetric private/public key) (recommended)

HMAC protocol

The HMAC protocol uses a shared secret key to both encrypt and verify the encrypted data. In our application, the authorization server encrypts the JWT signature with the secret key.

We copy the secret key in the *key* column of the *apps* table, along with the encryption algorithm in column *algorithm* (see *algorithms*). Bam-server can then verify token signatures using the secret key.

> **Attention:** The secret key is then as safe as your database content is. An attacker obtaining the key can not only forge queries for the bam-server, but for all other services based on the same authorization server. We advise to use the RSA protocol whenever possible.

RSA protocol

The RSA protocol is an asymmetric algorithm that uses a private, secret key to encrypt some data, which can be then verified (not decrypted) using a shared public key.

In our application, the authorization server encrypts the JWT signature with its private key, and provides the public key.

We copy the public key to the bam-server database, in the *key* column of the *apps* table, along with the encryption algorithm in column *algorithm* (see *algorithms*). Bam-server can then verify token signatures using the public key.

RSA keys are not strings, but can be stored and shared as text in these common formats:

- PEM format:

```
-----BEGIN RSA PUBLIC KEY-----
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAK7ttYaE/1ldsb0OJQDQhhDWqwuFWIyt
xgYIJH1HYA4UpA/Nm24fERIA1xi2Pomep6VTnQ/ThFP5hn2NyITwCIsCAwEAAQ==
-----END RSA PUBLIC KEY-----
```

- CER format ("certificate"):

```
-----BEGIN CERTIFICATE-----
MIIC8jCCAdqgAwIBAgIJY0fhkAkJqc1YMA0GCSqGSIb3DQEBBQUAMCAxHjAcBgNV
...
-----END CERTIFICATE-----
```

Copy one of these into the *key* column of the *apps* table.

Important: Database TEXT fields can only hold one line of text. You must replace all line returns by `\n`, for instance:

```
-----BEGIN RSA PUBLIC KEY-----\nMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAK7ttYaE/
↪1ldsb0OJQDQhhDWqwuFWIyt\nxgYIJH1HYA4UpA/Nm24fERIA1xi2Pomep6VTnQ/
↪ThFP5hn2NyITwCIsCAwEAAQ==\n-----END RSA PUBLIC KEY-----
```

JWT example

```
# Header
{
  "typ": "JWT",
  "alg": "RS256"
}

# Payload
{
  "name": "myUsername77",
  "iss": "myAppname",
  "exp": 31490863741,
  "iat": 1490863741,
  "sub": ...,
  "aud": ...,
  ...
}

# Signature
```

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  <public key>,
  <private key>
)

# Final token
<Base64(Header)>.<Base64(Payload)>.<Base64(Signature)>
```

For more details on JWTs, see jwt.io.

Supported algorithms

The values in the first column are the ones that can be put in *apps.algorithm*.

algorithm	description
"HS256"	HMAC using SHA-256 algorithm
"HS384"	HMAC using SHA-384 algorithm
"HS512"	HMAC using SHA-512 algorithm
"RS256"	RSASSA using SHA-256 algorithm
"RS384"	RSASSA using SHA-384 algorithm
"S512"	RSASSA using SHA-512 algorithm

Local database

Here we describe the local bam-server database. It links together app, user and sample identifers to set individual permissions.

Schema

Table name	Description
apps	The client applications, that query the bam-server.
users	The individual users of an app, as identified by auth tokens coming from the app.
samples	A sample corresponds to one BAM file.
users_samples	The attribution of samples to users

Here is the complete schema creation script (for MySQL):

```
CREATE TABLE `apps` (
    `id` INTEGER(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    `iss` VARCHAR(255) NOT NULL,
    `key` TEXT NOT NULL,
    `algorithm` VARCHAR(255) DEFAULT 'RS256',
    `description` VARCHAR(255) DEFAULT NULL,
    `isActive` TINYINT(1) NOT NULL DEFAULT 1
);

CREATE TABLE `users` (
    `id` INTEGER(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    `app_id` INTEGER(11) NOT NULL,
    `username` VARCHAR(255) NOT NULL,
```

```
    `group` VARCHAR(255) DEFAULT NULL,
    `isActive` TINYINT(1) DEFAULT 1,
    `isAdmin` TINYINT(1) DEFAULT 0,
    FOREIGN KEY (`app_id`) REFERENCES `apps`(`id`)
);

CREATE TABLE `samples` (
    `id` INTEGER(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    `filename` VARCHAR(255) NOT NULL,
    `project` VARCHAR(255) DEFAULT NULL,
    `hash` VARCHAR(255) DEFAULT NULL,
    `description` VARCHAR(255) DEFAULT NULL,
    `isOndisk` TINYINT(1) DEFAULT NULL,
    `isActive` TINYINT(1) NOT NULL DEFAULT 1
);

CREATE TABLE `users_samples` (
    `id` INTEGER(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    `user_id` INTEGER(11) NOT NULL,
    `sample_id` INTEGER(11) NOT NULL,
    `isActive` TINYINT(1) NOT NULL DEFAULT 1,
    FOREIGN KEY (`user_id`) REFERENCES `users`(`id`),
    FOREIGN KEY (`sample_id`) REFERENCES `samples`(`id`)
);
```

Fields description

apps.iss	App identifier - the "iss" claim of JWTs.
apps.key	The token signature verification key (a public key for RSA, a shared secret for HMAC).
apps.algorithm	The signature algorithm (one of HS256,HS384,HS512,RS256,RS384,RS512).
apps.description	[optional] Some text to describe the app.
users.app_id	The app ID.
users.username	The user identifier - the one passed in the JWT to identify the requester.
users.group	[optional] The name of a group the user belong to [1].
users.isAdmin	Whether the user has administrator rights (can create or delete users, samples, etc.).
samples.name	The sample identifier - the one used in the API to query the corresponding BAM file.
samples.filename	The name of the corresponding BAM file in `env.BAM_PATH` (see Configuration).
samples.project	[optional] The name of a project the sample belongs to [1].
samples.hash	[optional] The hash of the BAM file [1].
samples.description	[optional] Some text to describe the sample.
samples.isOnDisk	[optional] To mark a file as not found on disk anymore [1].
users_samples.user_id	The user ID.
users_samples.sample_id	The sample ID.

[1] These fields are not used directly, but can be useful for automatic management tasks.

Some test data

For convenience, this generates 3 apps with different signature algorithms, 2 users (1 admin), 2 samples, and attributes samples to the users.

```
INSERT INTO `apps` VALUES
(1, 'https://jdelafon.eu.auth0.com/', '-----BEGIN RSA PUBLIC KEY-----
↪\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtWWKxPv9vsWdRR/hcmJF\nsQjjUrMs/
↪OsVstyNJXwmWuhl3lNIZwwEDoJbnE9IKPyizyNwbnB9FmJnClCboUeP\nbkuIrDM63+S+PtX/
↪SQ9YI5yDxz+88dRYT86WP23wcWMO3txV2GAu62RVGSl48ZJP\nSyu94NBIiZOO5oDJpWDInhZphiMQ3u/
↪rEwlVxVMt0CTTInfl4iX0sCtymD2y6M38\nVrQwHOzSddFrbI58t4Rfal4SttwdmXONRnj7mrgl5G6v7IHEa/
↪HOrlT1rSLOMBKz\nOfmZy+bdlt5zrx3Adfzgn1BC6DGlG3Y9QYMOPpXjbzRO3rv9Fl5bRJyn5Ih82Cey\ndQIDAQAB\n-
↪----END RSA PUBLIC KEY-----', 'RS256', 'Using a public certificate in .cer format',␣
↪1),
(2, 'testapp', '-----BEGIN RSA PUBLIC KEY-----
↪\nMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAK7ttYaE/1ldsb0OJQDQhhDWqwuFWIyt\nxgYIJH1HYA4UpA/
↪Nm24fERIA1xi2Pomep6VTnQ/ThFP5hn2NyITwCIsCAwEAAQ==\n-----END RSA PUBLIC KEY-----',
↪'RS256', 'Using a public key in .pem format', 1),
(3, 'testapp-hmac', 'secretHMACkey', 'HS256', 'Using a shared secret key', 1);

INSERT INTO `users`(`id`,`app_id`,`username`,`isActive`,`isAdmin`) VALUES
(1, 1, 'admin@test.com', 1, 1),
(2, 1, 'test@test.com', 1, 0),
(3, 2, 'test@test.com', 1, 0),
(4, 3, 'test@test.com', 1, 0);

INSERT INTO `samples`(`id`,`name`,`filename`,`isActive`) VALUES
(1, 'sample1', 'test1.bam', 1),
(2, 'sample2', 'test2.bam', 1);

INSERT INTO `users_samples`(`user_id`,`sample_id`) VALUES
(1, 1),(1, 2),(2, 1);
```

Management API

A mini REST API has also been added to facilitate the entry of apps, users, samples and users_samples attributions in the database.

Only registered users tagged as *isAdmin* in the database have access to these functions. Like BAM query routes, these expect an Authorization header with a signed Bearer token to identify the requester (see *Authorization protocol*).

Add/remove an app

To register an application to the bam-server, i.e. make bam-server understand (verify) signed tokens coming from the registered applications.

```
PUT /apps
```

Expects a JSON body of the type

```
{
  "iss": "my-app",
  "key": "secret",
  "description": ""
}
```

- **iss**: the issuer, the app identifier such as present in JWTs under the *iss* claim.
- **key**: the signature verification key. Either a shared HMAC secret or an RSA public certificate (.pem or .cer formats, replacing newlines by `\n`).

- **description**: [optional] a description of the app.

```
DELETE /apps
```

Expects a JSON body of the type

```
{
  "iss": "my-app",
}
```

Note: One cannot add an app that already exists (same **iss**).

Add/remove users

To register users that can make BAM queries. Users will be given the same *appId* as the admin inserting them.

```
PUT /users
```

Expects a JSON body of the type

```
{
  "users": [
    {"username": "user1"},
    {"username": "user2"}
  ]
}
```

```
DELETE /users
```

Expects a JSON body of the type

```
{
  "users": ["user1", "user2"],
}
```

Note: One cannot add users that already exist, or delete users that do not exist. If that happens for one of the users of the query, nothing is inserted or deleted at all.

Add/remove samples

To register BAM files available for query.

```
PUT /samples
```

Expects a JSON body of the type

```
{
  "samples": [
    {
      "name": "A",
      "filename": "/"
```

```
    },
    {
      "name": "B",
      "filename": "/"
    }
  ]
}
```

- **name**: the sample identifer, such as used in the BAM query API.
- **filename**: file name, or path to the BAM file relatively to the configured *BAM_PATH*.

```
DELETE /samples
```

Expects a JSON body of the type

```
{
  "samples": ["A", "B"]
}
```

Note: One cannot add samples that already exist, or delete samples that do not exist. If that happens for one of the samples of the query, nothing is inserted or deleted at all.

Add/remove an attribution

To give or revoke access of a certain user to a certain BAM file.

```
PUT /users_samples
DELETE /users_samples
```

Expect a JSON body of the type

```
{
  "users_samples": [
    {
      "sample": "S1",
      "username": "A"
    },
    {
      "sample": "S2",
      "username": "B"
    }
  ]
}
```

Note: All user identifiers and sample identifiers in a query must be found in the database. If it is not the case of one of them, nothing is inserted or deleted at all.

Implementation examples

Here are some practical solutions using external products that we have used with success in practice. We do not advertise for them but only provide these resources to illustrate concretely how bam-server can be used.

Using IGV.js as reads viewer

The IGV team has produced a Javascript version of its famous BAM viewer, available at https://github.com/igvteam/igv.js/tree/master.

All you need to do to use it is pass the right URLs to its `options.tracks` parameter:

```
options = {
    showNavigation: true,
    showRuler: true,
    genome: "hg19",
    locus: "chr1:761997-762551"
    tracks: [
        {
            url: 'http://localhost:9000/bam/range/'+ sampleName +'?token='+␣
↪AuthService.getToken(),
            indexURL: 'http://localhost:9000/bai/'+ sampleName +'?token='+␣
↪AuthService.getToken(),
            type: "alignment",
            name: sampleName + ' [Protected resource!!]',
            displayMode: 'SQUISHED',
            alignmentRowHeight: 4,
        }
    ]
};
```

Here we use

```
/bai/:sample?token=<token>
/bam/range/:sample?token=<token>
```

from the bam-server API to read respectively the BAM index and the BAM itself. IGV.js is able to parse the index to find which bytes range to ask for in a RANGE request.

Note: The IGV.js team is about to release a way to pass Authorization headers instead of having to pass the token in the URL (which can be unsafe with reusable tokens).

Result:

Example of using Auth0 as authorization server

Auth0 is a commercial solution that provides an easy-to-manage authorization server (for free if you have few users and minimal requirements) implementing the OAuth2 protocol. Its only role is to identify users of the client app and return signed tokens.

1. Create an Auth0 client and register it to bam-server
   (a) Create an Auth0 account
   (b) Create a new client app, say “bam-server”. It is given a Client ID and a Client secret.

Fig. 1.2: Demo interface using bam-server to display read alignments.

(c) Go to "Clients > Show advanced settings". In the "OAuth" tab, choose "RS256". This changes the encryption protocol from the default HMAC (symmetric, shared secret) to RSA (asymmetric private/public - more secure).

(d) Still in the advanced settings, copy the public certificate from "Certificates > Signing Certificate" in CER or PEM format.

(e) Create an app in the *apps* table of the bam-server database, with the "Domain name" in the *iss* column.

(f) Replace line returns in the public certificate by `\n` and paste this as a single line in the *key* column.

2. Obtain a token (for testing; the client app is out of this scope)

(a) Let's say your client web app is served as localhost:3000.

(b) Add "localhost:3000" to Auth0 "Clients > Allowed callback URLs".

(c) Point your browser to:

```
https://<domain>/authorize
    ?scope=openid name nickname email user_id
    &response_type=token
    &client_id=<client_id>
    &redirect_uri=localhost:3000
    &nonce=1234&state=1234
```

where you replace <*domain*> and <*client_id*> by your own Auth0 client settings. N.B. The "scope" argument is what permits to have the fields "name", "email", etc. in the token, which we can use a user identifier.

(d) Log in your client app; you get redirected to localhost:3000/ and you can find an "id_token" in the url. This is a JWT that bam-server accepts (*not* "access_token"). For instance:

```
http://localhost:3000?access_token=ZVvVzzBIucYdhnW3&expires_in=86400&id_
↪token=xxxxxxx.yyyyyyy.zzzzzzz&token_type=Bearer&state=1234
```

To decode the body of the token (which is public!), you can use jwt.io. Note the *iss* and *name* claims.

3. Create a user, a sample, a users_samples entry in the database

(a) Create a new user using the *name* from the JWT as *username*, and the ID of the app you created in 1. as *app_id*.

```
INSERT INTO users(app_id, username) VALUES (<app_id>, <username>);
```

(b) Add a sample:

```
INSERT INTO bam(sample, filename) VALUES (<sample_name>, <bam_filename>);
```

Make sure the BAM file is available at *env.BAM_PATH* (see Configuration in *Installation guidelines*).

(c) Attribute the sample to a user:

```
INSERT INTO users_bam(user_id, bam_id) VALUES (<user_id>, <bam_id>);
```

where <*user_id*> and <*bam_id*> are the respective ids of the rows inserted above.

4. Use the client app to query

Try to read a BAM index on behalf of the user you created (i.e. using the token we got in 2.):

curl -i -H "Authorization: Bearer xxxxxxx.yyyyyyy.zzzzzzz" http://localhost:9000/bai/SAMPLE1

CHAPTER 2

Indices and tables

- genindex
- modindex
- search