Varda Documentation

Release 0.1.0.dev

Martijn Vermaat <martijn@vermaat.name>

May 26, 2016

Contents

Warning: This is a work in progress, probably not yet ready for use!

Varda is an application for storing genomic variation data obtained from next-generation sequencing experiments, such as full-genome or exome sequencing of individuals or populations. Variants can be imported from standard formats such as VCF files and annotated with their frequencies in previously imported datasets.

Varda is implemented as a service exposing a RESTful HTTP interface. Two clients for this interface are under development:

- Manwë - Python client library and command line interface to Varda.
- Aulë - Web interface to Varda.

Please see the tutorial for how to get started with all of these components.

CHAPTER 1

User documentation

This section of the documentation aims to guide users through working with Varda.

1.1 Introduction

Varda is an application for storing genomic variation data obtained from next-generation sequencing experiments, such as full-genome or exome sequencing of individuals or populations. Variants can be imported from standard formats such as VCF files and annotated with their frequencies in previously imported datasets.

Varda is implemented as a service exposing a RESTful HTTP interface. Two clients for this interface are under development:

- Manwë - Python client library and command line interface to Varda.
- Aulë - Web interface to Varda.

1.1.1 Use cases

The following are some example use cases which Varda is designed to support.

- *Private exome variant database for a sequencing lab*

  Installed on the local network, Varda can be used to import and annotate variants from all exome sequencing experiments at a sequencing lab. Additionally, the database could contain public datasets from population studies (e.g., 1000 Genomes, Genome of the Netherlands), such that all exome experiments are also annotated with frequencies in those studies.

- *Shared database between several groups*

  Several sequencing centers can import their variants in a central Varda installation which can subsequently be used by the same centers for frequency annotation. The system can be setup such that annotation is only possible on previously imported data (to encourage sharing).

  Data from one center can only be accessed anonymized by other groups, since only the frequencies over the entire databased are available. To accomodate even stricter anonymity, samples can be imported after pooling.

- *Publicly sharing variant frequencies from a population study*

  Variation data from a population study can be imported in a Varda installation accessible over the internet such that others can annotate their data with frequencies in the study.

For contrast, consider the following examples of what Varda is *not* designed to do.

- *Sharing and browsing genomic variants*

  Varda is focussed on sharing variant frequencies only, and as such is not designed for direct browsing. Other systems, such as LOVD, are much more suitable for sharing and browsing genomic variants and additionally store phenotypes and other metadata.

- *Ad-hoc exploration of genomic variation*

  Again, Varda is focussed on sharing variant frequencies only, and does not store additional metadata nor does it allow for effective exploration of variants. If you have variation data from a disease or population study which you want to analyse in a flexible way, have a look at gemini.

1.1.2 Implementation

The server is implemented in Python using the Flask framework and directly interfaces the PostgreSQL (or MySQL) database backend using SQLAlchemy. It exposes a RESTful API over HTTP where response payloads are JSON-encoded.

Long-running actions are executed asynchonously through the Celery distributed task queue.

1.2 User guide

Todo

Structure this guide, it's an incomplete random collection of paragraphs.

1.2.1 Variant frequencies and genomic coverage

Todo.

1.2.2 User roles

Todo.

1.2.3 Sample activation

Todo.

1.2.4 Duplication of data

Todo.

Calculating checksums of all imported data is used as a measure to prevent the same data to be imported twice. Of course, this is quite a weak measure in that it can easily be circumvented, so its main value lies in preventing accidental duplicate imports.

Note: There is a high chance of checksum collision with empty files (e.g., no variants were called). A solution is to always have something unique to the sample in the header of the file.

1.2.5 A model for trading data

For certain use cases it may be desirable that variant frequencies can only be retrieved from Varda by annotating variants that are imported in Varda (see *Shared database between several groups*). In this model variant observations are traded for variant frequencies.

Varda facilitates this with the *trader* role. The *trader* role gives a user permission to annotate a data source, but only if that data source has been imported as part of an active sample.

See *User roles* for more information on roles.

1.2.6 Sample anonymity

Todo. Only global frequencies, except for public samples. Of course depending on the number of samples in the database. See *Pooling samples*.

1.2.7 Pooling samples

By design, Varda cannot be queried in a way to reconstruct the genotype for a specific sample, unless that sample is explicitely marked as being public (see *Sample anonymity*). However, the sample genotypes are stored in the Varda database and for various reasons you might not be comfortable with that. This can be addressed by *pooling* samples before sending them to Varda, a trick that loses individual genotypes at no cost in functionality.

The idea is to mix the data from several samples together and send the result to Varda as one sample. Variant frequency calculations are not affected by this, yet individual genotypes are irrevocably scrambled. There are different ways to mix variant calls from different samples, you can find some examples below.

Besides variant calls, coverage information could also be mixed. However, it is probably not worth the trouble since this is not sensitive data.

Note: The effect of pooling is related to the number of samples. The greater the pool size, the better it works.

Example: merge single-sample VCF files

Starting with a VCF file per sample, one can simply concatenate all of them (minus the headers) and sort the result by chromosomal position. The resulting file looks like a single-sample VCF file, just with many more variant calls in it.

```
$ (grep '^#' 1.vcf; grep -hv '^#' *.vcf | sort -k 1,1 -k 2n,2) > pooled.vcf
```

Example: shuffle a multi-sample VCF file

If you already have a multi-sample VCF file containing variant calls for your samples, you can randomize the sample columns repeatedly for each line. The resulting file has lost individual genotypes, but contains the same variant frequency information.

```
$ grep '^#' samples.vcf > shuffled.vcf
$ paste \
    <(grep -v '^#' samples.vcf | cut -f 1-9) \
    <(grep -v '^#' samples.vcf | cut -f 10- \
      | xargs -L 1 bash -c 'shuf -e $* | paste -s' _) \
    >> shuffled.vcf
```

CHAPTER 2

Managing Varda

Start here if you're responsible for getting Varda running on a system.

2.1 Installation

Varda depends on a database server, a message broker, a task result backend, Python 2.7, and several Python packages. This section walks you through installing Varda using PostgreSQL as database server and Redis as message broker and task result backend, which is the recommended setup.

Note: All operating system specific instructions assume installation on a Debian 7 *wheezy* system. You'll have to figure out the necessary adjustements yourself if you're on another system.

The following steps will get Varda running on your system with the recommended setup:

- *Database server: PostgreSQL*
- *Message broker and task result backend: Redis*
- *Python virtual environment*
- *Varda setup*

At the bottom of this page some *alternative setups* are documented.

2.1.1 If you're in a hurry

The impatient can install and run Varda without a database server and more such nonsense with the following steps:

```
$ pip install -r requirements.txt
$ python -m varda.commands debugserver --setup
```

Don't use this for anything serious though.

2.1.2 Database server: PostgreSQL

Install PostgreSQL and add a user for Varda. Create a database (e.g. `varda`) owned by the new user. For example:

```
$ sudo apt-get install postgresql
$ sudo -u postgres createuser --superuser $USER
$ createuser --pwprompt --encrypted --no-adduser --no-createdb --no-createrole varda
$ createdb --encoding=UNICODE --owner=varda varda
```

```
```

Also install some development libraries needed for building the `psycopg2` Python package later and add the package to the list of requirements:

```
$ sudo apt-get install python-dev libpq-dev
$ echo psycopg2 >> requirements.txt
```

This will make sure the Python PostgreSQL database adapter gets installed in the *Python virtual environment* section.

See also:

Database server: MySQL Alternatively, MySQL can be used as database server.

Database server: SQLite Alternatively, SQLite can be used as database server.

Dialects – SQLAlchemy documentation In theory, any database supported by SQLAlchemy could work.

2.1.3 Message broker and task result backend: Redis

Varda uses Celery for distributing long-running tasks. A message broker is needed for communication between the server process and worker processes. Simply install Redis and you're done.

```
$ sudo apt-get install redis-server
```

See also:

Message broker: RabbitMQ Alternatively, RabbitMQ can be used as message broker.

Brokers – Celery documentation It should be possible to use any message broker and any task result backend supported by Celery.

2.1.4 Python virtual environment

It is recommended to run Varda from a Python virtual environment, using virtualenv. Installing virtualenv and creating virtual environment is not covered here.

Assuming you created and activated a virtual environment for Varda, install all required Python packages:

```
$ pip install -r requirements.txt
```

Now might be a good idea to run the unit tests:

```
$ nosetests -v
```

If everything's okay, install Varda:

```
$ python setup.py install
```

See also:

virtualenv `virtualenv` is a tool to create isolated Python environments.

virtualenvwrapper `virtualenvwrapper` is a set of extensions to the `virtualenv` tool. The extensions include wrappers for creating and deleting virtual environments and otherwise managing your development workflow.

2.1.5 Varda setup

Varda looks for its configuration in the file specified by the `VARDA_SETTINGS` environment variable. First create the file with your configuration settings, for example:

```
$ export VARDA_SETTINGS=~/varda/settings.py
$ cat > $VARDA_SETTINGS
DATA_DIR = '/data/varda'
SQLALCHEMY_DATABASE_URI = 'postgresql://varda:*****@localhost/varda'
BROKER_URL = 'redis://'
CELERY_RESULT_BACKEND = 'redis://'
```

Make sure `DATA_DIR` refers to a directory that is writable for Varda. This is where Varda stores uploaded and generated files.

A script is included to setup the database tables and add an administrator user:

```
$ varda setup
```

You can now proceed to *Running Varda*.

See also:

Configuration For more information on the available configuration settings.

2.1.6 Alternative setups

The remainder of this page documents some alternatives to the recommended setup documented above.

Database server: MySQL

Install MySQL and create a database (e.g. `varda`) with all privileges for the Varda user. For example:

```
$ sudo apt-get install mysql-server
$ mysql -h localhost -u root -p
> create database varda;
> grant all privileges on varda.* to varda@localhost identified by '*****';
```

Also install some development libraries needed for building the `MySQL-python` Python package later and add the package to the list of requirements:

```
$ sudo apt-get install python-dev libmysqlclient-dev
$ echo MySQL-python >> requirements.txt
```

This will make sure the Python MySQL database adapter gets installed in the *Python virtual environment* section.

See also:

Database server: PostgreSQL The recommended setup uses PostgreSQL as database server.

Database server: SQLite

You probably already have all you need for using SQLite.

See also:

Database server: PostgreSQL The recommended setup uses PostgreSQL as database server.

Message broker: RabbitMQ

Preferably install RabbitMQ from the APT repository provided by RabbitMQ. Example:

```
$ sudo apt-get install rabbitmq-server
$ sudo rabbitmqctl add_user varda varda
$ sudo rabbitmqctl add_vhost varda
$ sudo rabbitmqctl set_permissions -p varda varda '.*' '.*' '.*'
```

See also:

Message broker and task result backend: Redis The recommended setup uses Redis as message broker.

2.2 Upgrading

Before upgrading Varda, stop the currently running server and Celery workers. Then, update your copy of the source code (using for example `git pull` on an existing git clone).

Make sure to install any new requirements:

```
$ pip install -r requirements.txt
```

Now install the new version:

```
$ python setup.py install
```

Managing database migrations is done using Alembic. This command will move your database to the latest schema:

```
$ alembic upgrade head
```

You can now restart the server and Celery workers.

2.3 Configuration

This section describes how to configure Varda and includes a list of all available configuration settings.

Varda looks for its configuration in the file specified by the `VARDA_SETTINGS` environment variable. Make sure to always have this environment variable set when invoking any component of Varda. One way of doing this is by exporting it:

```
$ export VARDA_SETTINGS=~/varda/settings.py
```

If you like, you can add this command to your `~/.bashrc` to have it executed every time you open a shell.

Another way is by prefixing your invocations with `VARDA_SETTINGS=...`. For example:

```
$ VARDA_SETTINGS=~/varda/settings.py varda debugserver
```

2.3.1 Example configuration

If you followed the steps in *Installation*, this is a standard configuration file that will work for you:

```
DATA_DIR = '/data/varda'
SQLALCHEMY_DATABASE_URI = 'postgresql://varda:*****@localhost/varda'
BROKER_URL = 'redis://'
CELERY_RESULT_BACKEND = 'redis://'
```

This is not yet a minimal configuration. In fact, you can run Varda without a configuration file since the default configuration works out of the box. The default configuration uses an in-memory database, broker, and task result backend and a temporary directory for file storage, so it is not recommended for anything more than playing around.

The next section describes all available configuration settings.

2.3.2 Configuration settings

Note that the configuration file is interpreted as a Python module, so you can use arbitrary Python expressions as configuration values, or even import other modules in it.

Unsetting a configuration setting is done by using the value *None*. If no default value is mentioned for any configuration setting below it means it is not set by default.

HTTP server settings

API_URL_PREFIX URL prefix to serve the Varda server API under.

MAX_CONTENT_LENGTH Maximum size for uploaded files.

*Default value: 1024**3* (1 gigabyte)

CORS_ALLOW_ORIGIN A URI (or $\star$) that may access resources via cross-origin resource sharing (CORS), used in the Access-Control-Allow-Origin response header.

Default value: None

Data files settings

DATA_DIR Directory to store files (uploaded and generated).

Default value: tempfile.mkdtemp() (a temporary directory)

SECONDARY_DATA_DIR Secondary directory to use files from, for example uploaded there by other means such as SFTP (Varda will never write there, only symlink to it).

SECONDARY_DATA_BY_USER Have a subdirectory per user in SECONDARY_DATA_DIR (same as user login).

Default value: False

Reference genome settings

GENOME

Location of reference genome Fasta file.

Varda can use a reference genome to check and normalize variant descriptions. Specify the location to a FASTA file with the `GENOME` setting in the configuration file:

```
$ cat >> $VARDA_SETTINGS
GENOME = '/usr/local/genomes/hg19.fa'
REFERENCE_MISMATCH_ABORT = True
```

A Samtools "faidx" compatible index file will automatically be created if it does not exist yet.

REFERENCE_MISMATCH_ABORT Abort entire task if a reference mismatch occurs.

Default value: True

Database settings

SQLALCHEMY_DATABASE_URI SQLAlchemy database connection URI specifying the database used to store users, samples, variants, etcetera.

Database system	Example URI
PostgreSQL	`postgresql://user:*****@localhost/varda`
MySQL	`mysql://user:*****@localhost/varda`
SQLite	`sqlite:///varda.db`

See the SQLAlchemy documentation on Engine Configuration for more information.

Default value: `sqlite://` (in-memory SQLite database)

Celery settings

The most relevant configuration settings for varda relating to Celery are described here, but many more are available. See the Celery documentation on Configuration and defaults for information on all available configuration settings.

BROKER_URL Message broker connection URL used by Celery.

Broker system	Example URI
Redis	`redis://`
RabbitMQ	`amqp://varda:*****@localhost:5672/varda`

See the Celery documentation on Broker settings for more information.

Default value: `memory://`

CELERY_RESULT_BACKEND Task result backend used by Celery.

Backend system	
Redis	`redis://`
Database using SQLAlchemy	`database`
memcached	`cache`

Default value: `cache`

See the Celery documentation on Task result backend settings for more information.

CELERY_RESULT_DBURI SQLAlchemy database connection URI specifying the database used by Celery as task result backend if *CELERY_RESULT_BACKEND* is set to `database`.

CELERY_CACHE_BACKEND memcached connection URI specifying the server(s) used by Celery as task result backend if *CELERY_RESULT_BACKEND* is set to `cache`.

Default value: `memory` (no server, stored in memory only)

CELERYD_LOG_FILE Location of Celery log file.

CELERYD_HIJACK_ROOT_LOGGER Todo: Look into this setting.

Miscellaneous settings

TESTING If set to *True*, Varda assumes to be running its unit tests. This is done automatically in the provided test suite, so you should never have to change this setting.

Default value: False

2.4 Running Varda

Varda comes with a built-in test server that's useful for development and debugging purposes. You can start it like this:

```
$ varda debugserver
 * Running on http://127.0.0.1:5000/
```

You can now point your webbrowser to the URL that is printed and see a json- encoded status page.

This won't get you far in production though and there are many other possibilities for deploying Varda. Recommended is the Gunicorn WSGI HTTP server, which you could use like this:

```
$ gunicorn varda:create_app\(\) -w 4 -t 600 --max-requests=1000
```

See the Gunicorn website for documentation.

Varda distributes long-running tasks (such as importing and annotating variant files) using Celery. For running such tasks, you have to start at least one Celery worker node:

```
$ celery worker -A varda.worker.celery -l info --maxtasksperchild=4

 -------------- celery@hue v3.0.17 (Chiastic Slide)
---- **** -----
--- * ***  * -- [Configuration]
-- * - **** --- . broker:      redis://localhost:6379//
- ** ---------- . app:         varda:0x3602c50
- ** ---------- . concurrency: 8 (processes)
- ** ---------- . events:      OFF (enable -E to monitor this worker)
- ** ----------
- *** --- * --- [Queues]
-- ******* ---- . celery:      exchange:celery(direct) binding:celery
--- ***** -----

[Tasks]
  . varda.tasks.import_coverage
  . varda.tasks.import_variation
  . varda.tasks.ping
  . varda.tasks.write_annotation

[2013-04-05 17:39:59,882: WARNING/MainProcess] celery@hue ready.
[2013-04-05 17:39:59,886: INFO/MainProcess] consumer: Connected to redis://localhost:6379//.
```

CHAPTER 3

Tutorial

The tutorial shows you how to setup Varda with the Aulë web interface and Manwë command line client, and how to import and query an example dataset.

3.1 Tutorial

This tutorial shows you how to setup Varda with the Aulë web interface and Manwë command line client, and how to import and query an example dataset.

The example dataset is taken from the Varda unit tests and is limited to the first 200,000 bases of human chromosome 20 (GRCh37/hg19).

3.1.1 Setting up Varda

Follow the *installation instructions* to install Varda. Configure Varda to use `hg19.fa` in the `tests/data` directory as reference genome and enable cross-origin resource sharing (CORS) (this allows Aulë to communicate with Varda). The Varda configuration file may look something like this:

```
DATA_DIR = 'data'
SQLALCHEMY_DATABASE_URI = 'sqlite:///varda.db'
BROKER_URL = 'redis://'
CELERY_RESULT_BACKEND = 'redis://'
GENOME = 'tests/data/hg19.fa'
CORS_ALLOW_ORIGIN = '*'
```

Remember to point the `VARDA_SETTINGS` environment variable to the configuration file before continuing.

See also:

Configuration More information on available configuration settings.

Start Varda and a Celery worker node as described in *Running Varda*:

```
$ varda debugserver
```

and:

```
$ celery worker -A varda.worker.celery -l info
```

Opening http://127.0.0.1:5000/genome in your browser should now show you a JSON representation of the reference genome configuration.

3.1.2 Setting up Aulë

Get the source code for Aulë, configure it to use MyGene.info with GRCh37/hg19, and run it:

```
$ git clone https://github.com/varda/aule.git
$ cd aule
$ nano config.js
AULE_CONFIG = {
  BASE: '/',
  API_ROOT: 'http://127.0.0.1:5000/',
  PAGE_SIZE: 50,
  MANY_PAGES: 13,
  MY_GENE_INFO: {
    species: 'human',
    exons_field: 'exons_hg19'
  }
  MY_GENE_INFO: null
};
$ npm install
$ npm run dev
```

You can now open http://localhost:8000/ in your browser, which should show you the Aulë homepage. Login with `admin` and the password you choose during Varda setup.

3.1.3 Setting up Manwë

Manwë authenticates with the Varda API using a token. You can generate a token in the Aulë web interface by choosing *API tokens* in the menu and clicking *Generate API token*. Copy the token by clicking *Show token*.

Install Manwë and create a configuration file with the token you just created:

```
$ pip install manwe
$ nano manwe.cfg
API_ROOT = 'http://127.0.0.1:5000'
TOKEN = 'c7fa8780025c8efa5077567434e0fcb56274fbb0'
```

Verify that everything is setup correctly by listing all Varda users:

```
$ manwe users list -c manwe.cfg
User:   /users/1
Name:   Admin User
Login:  admin
Roles:  admin
```

Note: Instead of including `-c manwe.cfg` in every invocation, you can also copy this file to `~/.config/manwe/config` (`config` should be the name of the file) where Manwë will pick it up automatically.

3.1.4 Importing exome sequencing data

Let's import an example set of variant calls from an exome sequencing experiment. The file `tests/data/exome.vcf` contains some variant calls on chromosome 20 for one individual and `tests/data/exome.vcf` contains regions on chromosome 20 where the sequencing was deep enough (or of high enough quality) to do variant calling:

```
$ cat tests/data/exome.vcf
##fileformat=VCFv4.1
##samtoolsVersion=0.1.16 (r963:234)
...
#CHROM  POS     ID  REF    ALT  QUAL  FILTER  INFO  FORMAT    -
chr20   76962   .   T      C    173   .       ...   GT:PL:GQ  0/1:203,0,221:99
chr20   126159  .   ACAAA  A    217   .       ...   GT:PL:GQ  0/1:255,0,255:99
chr20   126313  .   CCC    C    126   .       ...   GT:PL:GQ  0/1:164,250,0:99
...
$ cat tests/data/exome.bed
chr206811268631
chr207658177410
chr209002590400
...
```

Note: For any real data you import, it is best to always include both the variant calls in VCF format *and* a BED file of regions to include. This makes it possible for Varda to calculate accurate variant frequencies, also on regions that are not covered by some experiments.

Import the data as follows:

```
$ manwe samples import --vcf tests/data/exome.vcf --bed tests/data/exome.bed \
>     -l -w 'Exome sample'
Added sample: /samples/1
Added data source: /data_sources/1
Started variation import: /variations/1
Added data source: /data_sources/2
Started coverage import: /coverages/1
[##################################] 100/100 - 00:00:02
Imported variations and coverages for sample: /samples/1
```

Note: The `-l` argument instructs Varda to use the `PL` column instead of the `GT` column to derive the genotypes. Use it when variant calling was done with Samtools.

Since Varda supports importing data for a sample in multiple steps, new samples are inactive by default to prevent using them in frequency calculations until everything is complete. Activate the sample you just imported with:

```
$ manwe samples activate /samples/1
Activated sample: /samples/1
```

If you go back to the Aulë web interface and choose *Samples* in the menu, you should see the exome sample you just imported.

3.1.5 Importing aggregate data from 1000 Genomes

Sometimes it makes sense to calculate variant frequencies within a dataset separately, as opposed to global frequencies over all datasets. An example might be a large public population study such as the 1000 Genomes project. Varda allows you to import a dataset like this without providing coverage data (i.e., the BED file).

The `tests/data/1kg.vcf` file contains a subset of variant calls from the 1000 Genomes project over 1092 individuals. Import it as follows:

```
$ manwe samples import --vcf ../varda/tests/data/1kg.vcf -s 1092 -p \
>     --no-coverage-profile -w '1000 Genomes'
Added sample: /samples/2
Added data source: /data_sources/3
```

```
Started variation import: /variations/2
[##################################] 100/100 - 00:00:02
Imported variations and coverages for sample: /samples/2
$ manwe samples activate /samples/2
Activated sample: /samples/2
```

Note: Samples imported without coverage profile are automatically excluded from global variant frequency calculations. Instead, they may be queried separately.

3.1.6 Querying variant frequencies

Aulë allows for some ad-hoc querying of variant frequencies globally and per sample, as well as by variant, by region and by transcript region. Choose *By region* in the menu and set:

Query: *Global query*

Chromosome: *chr20*

Region begin: *1*

Region end: *200000*

This should show you the variants from the exome sequencing example, all with frequency 1.0 and *N=1* (since it's the only sample used in the calculation).

You can run the same query on the 1000 Genomes data by setting:

Query: *Sample query (1000 Genomes)*

As an alternative to setting the region manually, you can also choose *By transcript* in the menu and select a region based on a gene transcript. The exome example has two variants in the DEFB126 gene. You can select it by clicking on *Choose a transcript* and typing `DEFB126`.

3.1.7 Annotating variants

The ad-hoc frequency queries with Aulë are nice for one-time lookups, but you would presumably also want to automate this on a larger scale. Manwë allows you to annotate local VCF or BED files with variant frequencies by supplying a list of queries:

```
$ manwe annotate-vcf -q GLOBAL '*' -q 1KG 'sample:/samples/2' -w \
>     tests/data/exome.vcf
Added data source: /data_sources/4
Started annotation: /annotations/1
[##################################] 100/100 - 00:00:02
Annotated VCF file: /data_sources/5
$ manwe data-sources download /data_sources/5 > exome.annotated.vcf.gz
```

The resulting VCF file is annotated with several fields in the `INFO` column.

CHAPTER 4

REST API documentation

Developers of client applications can read how to communicate with the Varda REST API in this section.

4.1 Introduction to the REST API

For communication with client applications, Varda exposes an API *following the REST architectural style*. The API represents resources in JSON format and user authentication is done using HTTP Basic Authentication.

Start by going through the *API manual*. After that, read through the documentation for the individual *resources*.

4.1.1 Conformance with REST

Although Varda tries to follow REST in its API, there are certainly parts of the API that are not completely in the spirit of REST.

Todo: More text here, at least covering the following points:

- JSON is not a hypertext format
- Accepting a request body with GET
- Todo

4.2 REST API manual

This page documents the REST server API exposed by Varda to client applications.

For more detailed information on specific API endpoints, see *REST API resources*.

4.2.1 An example request using *curl*

To get us started, here's an example of creating a new *sample* resource named *1000 Genomes* using *curl*:

```
curl -u user:password -X POST -H 'Content-Type: application/json' \
  -d '{"name": "1000 Genomes"}' https://example.com/samples/
```

The first thing to observe is that the request is authenticated using HTTP Basic Authentication (the `-u user:password` argument). See *Authentication* for more information.

The request body is a JSON document (specified with the *Content-Type* header) consisting of only a *name* field with value `1000 Genomes`. See *Passing data with a request* for more ways of sending data.

Finally, the request is done at the collection endpoint for *sample* resources using the *POST* method. This is the typical way of creating new resources and you can find more information on specific resources in *REST API resources*.

What we'll get back is the following HTTP response:

```
HTTP/1.1 201 CREATED
Server: gunicorn/18.0
Date: Sat, 16 Nov 2013 10:03:17 GMT
Connection: close
Content-Type: application/json
Content-Length: 282
Location: https://example.com/samples/140
Api-Version: 0.3.0

{
  "sample": {
    "uri": "/samples/140",
    "active": false,
    "added": "2013-11-16T10:47:28.711076",
    "coverage_profile": true,
    "name": "1000 Genomes",
    "notes": null,
    "pool_size": 1,
    "public": false,
    "user": {
      "uri": "/users/1"
    }
  }
}
```

The response body contains a representation of the created resource as a JSON document. We can follow the *Location* header to that same resource.

Note: For brevity, we'll omit many of the headers in example HTTP requests and responses from now on.

4.2.2 Authentication

Many requests require user authentication which can be provided with HTTP Basic Authentication or token authentication.

Authentication state can be checked on the *authentication* resource.

HTTP Basic Authentication

For interactive use of the API, the most obvious way of authenticating is by providing a username and password with HTTP Basic Authentication.

See also:

Wikipedia article on HTTP Basic Authentication

Token authentication

Automated communication with the API is better authenticated with a *token*. An authentication token is a secret string uniquely identifying a user that can be used in the *Authorization* request header. The value of this header should then be the string `Token`, followed by a space, followed by the token string. For example:

```
GET /samples HTTP/1.1
Authorization: Token 5431792000be7601697fb5a4005984ebdd60320c
```

Authentication tokens are themselves resources and can be managed using the API, see *Tokens*.

4.2.3 Passing data with a request

Data can be attached to a request in three ways:

1. As query string parameters.
2. As HTTP form data.
3. In a JSON-encoded request body.

Generally, using a JSON-encoded request body is preferred since it offers richer structure. For example, JSON has separate datatypes for strings and numbers, and supports nesting for more complex documents.

Note: A JSON-encoded request body is also accepted with GET requests, even though this is perhaps not true to the HTTP specification.

JSON-encoded bodies must always be accompanied with a `application/json` value for the *Content-Type* header.

String encoding of lists and objects

There is limited support for sending structured data as query string parameters or HTTP form data by serializing them. Lists are serialized by concatenating their items with `,` (comma) in between. Objects of name/value pairs are serialized similarly where the items are concatenations of name, `:` (colon) and value.

For example, the JSON list

```
[45, 3, 11, 89]
```

is serialized as:

```
45,3,11,89
```

Similarly, the JSON object

```
{
  "name1": "value1",
  "name2": "value2",
  "name3": "value3"
}
```

is serialized as:

```
name1:value1,name2:value2,name3:value3
```

Note: The decoding of these serializations is very primitive. For example, escaping of `,` (comma) or `:` (colon) is not possible.

4.2.4 Date and time

All date and time values are formatted as strings following ISO 8601.

See also:

Wikipedia article on ISO 8601

4.2.5 Queries

A *query* defines a set of samples, used to calculate observation frequencies over when annotationg variants. A query is represented as an object with two fields:

name (*string*) Name for this query (alphanumeric).

expression (*string*) Search query string.

The *expression* field is a boolean search query string in which clauses can reference *sample* resources and *group* resources. This is the grammar for query expressions:

```
<expression> ::= <tautology>
               | <clause>
               | "(" <expression> ")"
               | "not" <expression>
               | <expression> "and" <expression>
               | <expression> "or" <expression>

<tautology> ::= "*"

<clause> ::= <resource-type> ":" <uri>

<resource-type> ::= "sample" | "group"
```

The tautology query `*` matches all samples. A clause of the form `sample:<uri>` matches the sample with the given URI. A clause of the form `group:<uri>` matches samples that are in the group with the given URI.

When creating the set of samples matched by a query expression, only active samples with a coverage profile are considered. The exception to this are expressions of the form `sample:<uri>`, which can match inactive samples or samples without coverage profile.

As an example, the following is an expression matching the sample with URI `/samples/5` and samples that are in the group with URI `/groups/3` but not in the group with URI `/groups/17`:

```
sample:/samples/5 or (group:/groups/3 and not group:/groups/17)
```

4.2.6 Linked resources and embeddings

Resources can have links to other resources. In the resource representation, such a link is an object with a *uri* field containing the linked resource URI.

For some links, the complete representation of the linked resource can be embedded instead of just the *uri* field. This is documented with the resource representation.

For example, *sample* resources can have the linked *user* resource embedded:

```
GET /samples/130?embed=user
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "sample": {
    "uri": "/samples/130",
    "active": false,
    "added": "2013-03-30T00:18:48.298526",
    "coverage_profile": false,
    "name": "1KG phase1 integrated call set",
    "notes": null,
    "pool_size": 1092,
    "public": true,
    "user": {
      "uri": "/users/2",
      "added": "2012-11-30T20:28:11.409536",
      "email": null,
      "login": "martijn",
      "name": "Martijn Vermaat",
      "roles": [
        "trader",
        "annotator"
      ]
    }
  }
}
```

4.2.7 Collection resources

A collection resource is a grouping of any number of instance resources. Use a *POST* request on the collection resource to add an instance resource to it. Listing the instance resources is done with a *GET* request and comes with a number of utilities as described below.

Representation

A collection resource is represented as an object with two fields:

uri (*uri*) URI for this collection resource.

items (*list* of *object*) List of resource instances.

Range requests / pagination

A *GET* request on a collection resource **must** have a *Range* header specifying the range of instance resources (using *items* as range unit) that is requested. The response will contain the appropriate *Content-Range* header showing the actual range of instance resources that is returned together with the total number available.

Filtering

The returned list of recourse instances can sometimes be filtered by specifying values for resource fields. Documentation for the resource collection lists the fields that can be used to filter on.

For example, the *sample collection* resource can be filtered on the *public* and *user* fields.

Ordering

The ordering of the returned list of resource instances can be specified in the *order* field as a list of field names. Field names can be prefixed with a - (minus) for descending order or with a + (plus) for ascending order (default) and must be chosen from the documented set of orderable fields for the relevant collection resource.

For example, the *sample collection* resource can be ordered by the *name*, *pool_size*, *public*, *active*, and *added* fields.

All resource collections have a default order of their items which is usually ascending by URI (the *variant collection* being the exception).

Example *GET* request

We illustrate some of the described utilities by listing public samples ordered first descending by *pool_size* and second ascending by *name*. We request only the first 6 of them.

Example request:

```
GET /samples/?public=true&order=-pool_size,name HTTP/1.1
Range: items=0-5
```

Example response:

```
HTTP/1.1 206 PARTIAL CONTENT
Content-Type: application/json
Content-Range: items 0-5/8

{
  "sample_collection": {
    "uri": "/samples/",
    "items": [
      {
        "uri": "/samples/130",
        "name": "1KG phase1 integrated call set",
        "pool_size": 1092,
        "public": true,
        ...
      },
      {
        "uri": "/samples/134",
        "name": "My sample",
        "pool_size": 4,
        "public": true,
        ...
      },
      {
        "uri": "/samples/135",
        "name": "A new sample",
        "pool_size": 3,
        "public": true,
        ...
      },
      {
        "uri": "/samples/129",
        "name": "Another sample",
        "pool_size": 1,
        "public": true,
        ...
```

```
      },
      {
        "uri": "/samples/131",
        "name": "Sample 42",
        "pool_size": 1,
        "public": true,
        ...
      },
      {
        "uri": "/samples/128",
        "name": "Some test sample",
        "pool_size": 1,
        "public": true,
        ...
      }
    ]
  }
}
```

4.2.8 Tasked resources

A tasked resource is a type of resource associated with a server task. This task is submitted upon creation of a new resource instance (i.e., via a *POST* request on the corresponding collection resource).

Information on the server task can be obtained with a *GET* request on the instance resource. A task can be resubmitted by setting its *state* field to *submitted* in a *PATCH* request (this requires the *admin* role).

Representation

A tasked resource representation has a field *task* containing an object with the following fields:

state (*string*) Task state. Possible values for this field are *waiting*, *running*, *succes*, and *failure*.

progress (*integer*) Task progress as an integer in the range 0 to 100. Only present if the *state* field is set to *running*.

error (*object*) An *error object*. Only present if the *state* field is set to *failure*.

4.2.9 Versioning

The API is versioned following Semantic Versioning. Clients can (but are not required to) ask for specific versions of the API with a Semantic Versioning specification in the *Accept-Version* header.

If the server can match the specification, or *Accept-Version* is not set, the response will include the API version in the *Api-Version* header. If the specification cannot be matched, a 406 status is returned with a *no_acceptable_version* error code.

Example request with *Accept-Version* header, and corresponding response:

```
GET /
Accept-Version: >=0.3.1,<1.0.0
```

```
HTTP/1.1 200 OK
Api-Version: 0.4.2
```

Note: Currently the server implements one specific API version so there is no real negotiation on version. More sophisticated logic based on *Accept-Version* may be implemented in the future.

4.2.10 Error responses

If a request results in the occurrence of an error, the server responds by sending an appropriate HTTP status code and an error document containing:

1. An *error code* (*code*).
2. A human readable error message (*message*).

These fields are wrapped in an object called *error*.

Example request resulting in error

The following request aims to create a new *sample* resource with name *Test sample* and pool size *Thirty*:

```
POST /samples/ HTTP/1.1
Content-Type: application/json

{
  "name": "Test sample",
  "pool_size": "Thirty"
}
```

Of course, pool size should be encoded as an integer and therefore the following response is returned:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error":
    {
      "code": "bad_request",
      "message": "Invalid request content: value of field 'pool_size' must be of integer type"
    }
}
```

List of error codes

Here's an incomplete list of error codes with their meaning.

bad_request Invalid request content (*message* field contains more details).

basic_auth_required The request requires login/password authentication.

entity_too_large The request entity is too large.

forbidden Not allowed to make this request.

integrity_conflict The request could not be completed due to a conflict with the current state of the resource (*message* field contains more details).

internal_server_error The server encountered an unexpected condition which prevented it from fulfilling the request.

no_acceptable_version The requested version specification did not match an available API version.

not_found The requested entity could not be found.

not_implemented The functionality required to fulfill the request is currently not implemented.

unauthorized The request requires user authentication.

unsatisfiable_range Requested range not satisfiable.

4.2.11 Summary of HTTP status codes

We give a brief overview of response status codes sent by the server and their meaning. For more information, consult HTTP/1.1: Status Code Definitions.

200 Everything ok, the request has succeeded.

201 The request has been fulfilled and resulted in a new resource being created.

206 The server has fulfilled the partial GET request for the resource.

301 Moved permanently.

400 The request data was malformed.

401 The request requires user authentication.

403 Not allowed to make this request.

404 Nothing was found matching the request URI.

406 The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

409 The request could not be completed due to a conflict with the current state of the resource.

413 The request entity was too large.

416 Requested range not satisfiable.

500 Internal server error.

501 Not implemented.

4.3 REST API resources

4.3.1 API root

The API root resource contains links to top-level resources in addition to a server status code.

The root resource representation has the following fields:

uri (*uri*) URI for this resource.

status (*string*) Currently always `ok`, but future versions of the API might add other values (e.g. `maintanance`).

api_version (*string*) API version (see *Versioning*).

authentication (*object*) *Link* to the *authentication* resource.

genome (*object*) *Link* to the *genome* resource.

annotation_collection (*object*) *Link* to the *annotation collection* resource.

coverage_collection (*object*) *Link* to the *coverage collection* resource.

data_source_collection (*object*) *Link* to the *data_source collection* resource.

sample_collection (*object*) *Link* to the *sample collection* resource.

token_collection (*object*) *Link* to the *token collection* resource.

user_collection (*object*) *Link* to the *user collection* resource.

variant_collection (*object*) *Link* to the *variant collection* resource.

variation_collection (*object*) *Link* to the *variation collection* resource.

GET /

Returns the resource representation in the *root* field.

Example request:

```
GET / HTTP/1.1
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "root": {
    "uri": "/",
    "api_version": "0.3.0",
    "status": "ok",
    "authentication": {
      "uri": "/authentication"
    },
    "genome": {
      "uri": "/genome"
    },
    "annotation_collection": {
      "uri": "/annotations/"
    },
    "coverage_collection": {
      "uri": "/coverages/"
    },
    "data_source_collection": {
      "uri": "/data_sources/"
    },
    "sample_collection": {
      "uri": "/samples/"
    },
    "token_collection": {
      "uri": "/tokens/"
    },
    "user_collection": {
      "uri": "/users/"
    },
    "variant_collection": {
      "uri": "/variants/"
    },
    "variation_collection": {
      "uri": "/variations/"
    }
  }
}
```

4.3.2 Authentication

This resource reflects the current authentication state.

The authentication resource representation has the following fields:

uri (*uri*) URI for this resource.

authenticated (*boolean*) Whether or not the request is authenticated.

user (*object*) *Link* to a *user* resource if the request is authenticated, *null* otherwise.

GET /
Returns the resource representation in the *authentication* field.

Example request:

```
GET /authentication HTTP/1.1
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "authentication": {
    "uri": "/authentication",
    "authenticated": true,
    "user": {
      "uri": "/users/1"
      "added": "2012-11-30T20:14:27.954255",
      "email": null,
      "login": "admin",
      "name": "Admin User",
      "roles": [
        "admin"
      ],
    }
  }
}
```

4.3.3 Genome

If the server is configured with a reference genome, this resource lists its chromosomes.

The genome resource representation has the following fields:

uri (*uri*) URI for this resource.

chromosomes (*list* of *string*) List of chromosome names.

GET /
Returns the resource representation in the *genome* field.

4.3.4 Annotations

Annotation resources model sets of variants annotated with observation frequencies.

An annotation resource is a *tasked resource*. The associated server task is calculating frequencies on the linked original data source and writing the annotated data to the linked annotated data source.

An annotation is represented as an object with the following fields:

uri (*uri*) URI for this resource.

task (*object*) Task information, see *Tasked resources*.

original_data_source (*object*) *Link* to a *data source* resource (embeddable).

annotated_data_source (*object*) *Link* to a *data source* resource (embeddable).

Todo

Include and document the associated queries.

Collection

See also:

Collection resources

`GET /annotations/`
Returns a collection of annotations in the *annotation_collection* field.

Note: Requires having the *admin* role or being the user specified by the *user* filter.

Available filters:

- **user** (*uri*)

`POST /annotations/`
Adds an annotation resource.

Note: Requires having the *admin* role or being the owner of the data source specified by the *data_source* field.

Queries may have additional requirements depending on their expression:

1. Query expressions of the form `sample:<uri>` require one of the following:
   - Having the *admin* role.
   - Owning the sample specified by `<uri>`.
   - The sample specified by `<uri>` being public.
2. Query expressions of the form `*` require having the *admin*, *annotator*, or *trader* role, where the *trader* role additionally requires that *data_source* has been imported as variation in an active sample.
3. Query expressions containing only group clauses require the same as those of the form `*`, where the *annotator* and *trader* roles additionally require having the *group-querier* role.
4. Other query expressions require the same as those containing only group clauses, where the *annotator* and *trader* roles require having the *querier* role instead of the *group-querier* role.

Required request data:

- **data_source** (*uri*)

Accepted request data:

- **name** (*string*)
- **queries** (*list* of *object*)

Every object in the *queries* list defines a *query*; a set of samples over which observation frequencies are annotated. When annotating a VCF data source, any samples having this data source as variation are excluded.

Instances

GET /annotations/<id>
Returns the annotation representation in the *annotation* field.

Note: Requires having the *admin* role or being the owner of the annotation.

PATCH /annotations/<id>
Updates an annotation resource.

Note: Requires having the *admin* role.

Accepted request data:

- **task** (*object*)

4.3.5 Coverages

Coverage resources model sets of genomic regions having high enough coverage in sequencing to do variant calling.

A coverage resource is a *tasked resource*. The associated server task is importing the coverage data from the linked data source in the server database.

A coverage is represented as an object with the following fields:

uri (*uri*) URI for this resource.

task (*object*) Task information, see *Tasked resources*.

data_source (*object*) *Link* to a *data source* resource (embeddable).

sample (*object*) *Link* to a *sample* resource (embeddable).

Collection

See also:

Collection resources

GET /coverages/
Returns a collection of coverages in the *coverage_collection* field.

Note: Requires one or more of the following:

- Having the *admin* role.
- Being the owner of the sample specified by the *sample* filter.
- Setting the *sample* filter to a public sample.

Available filters:

- **sample** (*uri*)

POST /coverages/
Adds a coverage resource.

Note: Requires having the *admin* role or being the owner of the sample specified by the *sample* field.

Required request data:

- **data_source** (*uri*)
- **sample** (*uri*)

Instances

GET /coverages/<id>
Returns the coverage representation in the *coverage* field.

Note: Requires having the *admin* role or being the owner of the coverage.

PATCH /coverages/<id>
Updates a coverage resource.

Note: Requires having the *admin* role.

Accepted request data:

- **task** (*object*)

4.3.6 Data sources

Data source resources model data from files that are either uploaded to the server by the user or generated on the server.

The actual data is modeled by the *blob* subresource type.

A data source is represented as an object with the following fields:

uri (*uri*) URI for this resource.

added (*string*) Date and time this data source was added, see *Date and time*.

gzipped (*boolean*) Whether or not the data is compressed using gzip.

name (*string*) Human readable data source name.

filetype (*string*) Data filetype. Possible values for this field are *bed*, *vcf*, and *csv*.

data (*object*) *Link* to a *blob* resource.

user (*object*) *Link* to a *user* resource (embeddable).

Collection

See also:

Collection resources

GET /data_sources/
Returns a collection of data sources in the *data_source_collection* field.

Note: Requires having the *admin* role or being the user specified by the *user* filter.

Available filters:

- **user** (*uri*)

Orderable by: *name*, *filetype*, *added*

POST /data_sources/
Adds a data source resource.

Note: Requires *user authentication*.

Required request data:

- **name** (*string*)
- **filetype** (*string*)

Accepted request data:

- **gzipped** (*boolean*)
- **local_file** (*string*)
- **data** (*file*)

Instances

GET /data_sources/<id>
Returns the data source representation in the *data_source* field.

Note: Requires having the *admin* role or being the owner of the data source.

PATCH /data_sources/<id>
Updates a data source resource.

Note: Requires having the *admin* role or being the owner of the data source.

Accepted request data:

- **name** (*string*)

Blobs

GET /data_sources/<id>/data
Returns the gzipped data source data.

Warning: The response body will not be a JSON document.

Note: Requires having the *admin* role or being the owner of the data source.

4.3.7 Groups

Group resources model sample groups (e.g., disease type).

A group is representend as an object with the following fields:

uri (*uri*) URI for this resource.

name (*string*) Human readable group name.

Collection

See also:

Collection resources

`GET /groups/`
Returns a colleciton of groups in the *group_collection* field.

`POST /groups/`
Adds a group resource.

> **Note:** Requires having the *admin* or *importer* role.

Instances

`GET /groups/<id>`
Returns the group representation in the *group* field.

`PATCH /groups/<id>`
Updates a group resource.

> **Note:** Requires having the *admin* role.

4.3.8 Samples

Sample resources model biological samples which can contain one or more individuals.

A sample is represented as an object with the following fields:

uri (*uri*) URI for this resource.

active (*boolean*) Whether or not this sample is active.

added (*string*) Date and time this sample was added, see *Date and time*.

coverage_profile (*boolean*) Whether or not this sample has a coverage profile.

name (*string*) Human readable sample name.

notes (*string*) Human readable notes in Markdown format.

pool_size (*integer*) Number of individuals in this sample.

public (*boolean*) Whether or not this sample is public.

user (*object*) *Link* to a *user* resource (embeddable).

groups (*list* of *object*) *Links* to *group* resources (embeddable).

Collection

See also:

Collection resources

GET /samples/
Returns a collection of samples in the *sample_collection* field.

Note: Requires one or more of the following:

•Having the *admin* role.

•Being the user specified by the *user* filter.

•Setting the *public* filter to *True*.

Available filters:

•**groups** (*uri*)

•**public** (*boolean*)

•**user** (*uri*)

Orderable by: *name*, *pool_size*, *public*, *active*, *added*

POST /samples/
Adds a sample resource.

Note: Requires having the *admin* or *importer* role.

Required request data:

•**name** (*string*)

Accepted request data:

•**coverage_profile** (*boolean*)

•**groups** (*list* of *uri*)

•**notes** (*string*)

•**pool_size** (*integer*)

•**public** (*boolean*)

Instances

GET /samples/<id>
Returns the sample representation in the *sample* field.

Note: Requires one or more of the following:

•Having the *admin* role.

•Being the owner of the sample.

•The sample is public.

PATCH /samples/<id>
Updates a sample resource.

Note: Requires having the *admin* role or being the owner of the sample.

Accepted request data:

- **active** (*boolean*)
- **coverage_profile** (*boolean*)
- **groups** (*list* of *uri*)
- **name** (*string*)
- **notes** (*string*)
- **pool_size** (*integer*)
- **public** (*boolean*)

4.3.9 Tokens

Token resources model *authentication tokens* for API users.

A token is represented as an object with the following fields:

uri (*uri*) URI for this resource.

added (*string*) Date and time this sample was added, see *Date and time*.

key (*string*) Token key used for authentication.

name (*string*) Human readable sample name.

user (*object*) *Link* to a *user* resource (embeddable).

Collection

See also:

Collection resources

GET /tokens/
Returns a collection of tokens in the *collection* field.

Note: Requires having the *admin* role or being the user specified by the *user* filter.

Note: This request is only allowed using *HTTP Basic Authentication*, not token authentication.

Available filters:

- **user** (*uri*)

Orderable by: *name*, *added*

POST /tokens/
Adds a token resource.

Note: Requires having the *admin* or being the user specified by the *user* data field.

Note: This request is only allowed using *HTTP Basic Authentication*, not token authentication.

Required request data:

- **user** (*uri*)
- **name** (*string*)

Instances

GET /tokens/<id>
Returns the token representation in the *token* field.

Note: Requires having the *admin* role or being the owner of the token.

Note: This request is only allowed using *HTTP Basic Authentication*, not token authentication.

PATCH /tokens/<id>
Updates a token resource.

Note: Requires having the *admin* role or being the owner of the token.

Note: This request is only allowed using *HTTP Basic Authentication*, not token authentication.

Accepted request data:

- **name** (*string*)

4.3.10 Users

User resources model API users and their permissions.

A user is represented as an object with the following fields:

uri (*uri*) URI for this resource.

added (*string*) Date and time this user was added, see *Date and time*.

email (*string*) User e-mail address.

login (*string*) Login name used for authentication.

name (*string*) Human readable user name.

roles (*list* of *string*) Roles for this user. Possible values for this field are *admin*, *importer*, *annotator*, and *trader*.

Collection

See also:

Collection resources

GET /users/
Returns a collection of users in the *user_collection* field.

Note: Requires having the *admin* role.

Orderable by: *name*, *added*

POST /users/

Adds a user resource.

Note: Requires having the *admin* role.

Note: This request is only allowed using *HTTP Basic Authentication*, not token authentication.

Required request data:

- **login** (*string*)
- **password** (*string*)

Accepted request data:

- **name** (*string*)
- **email** (*string*)
- **roles** (*list* of *string*)

Instances

GET /users/<id>

Returns the user representation in the *user* field.

Note: Requires having the *admin* role or being the requested user.

PATCH /users/<id>

Updates a user resource.

Note: Requires having the *admin* role or being the requested user.

Note: This request is only allowed using *HTTP Basic Authentication*, not token authentication.

Accepted request data:

- **email** (*string*)
- **login** (*string*)
- **name** (*string*)
- **roles** (*list* of *string*)

4.3.11 Variants

Variant resources model genomic variants with their observed frequencies.

Note: The implementation of this resource is still in flux and it is therefore not documented.

A variant is represented as an object with the following fields:

uri (*uri*) URI for this resource.

Collection

See also:

Collection resources

`GET /variants/`
Returns a collection of variants in the *variant_collection* field.

`POST /variants/`
Adds a variant resource.

Instances

`GET /variants/<id>`
Returns the variant representation in the *variant* field.

4.3.12 Variations

Variation resources model sets of variant observations.

A variation resource is a *tasked resource*. The associated server task is importing the variation data from the linked data source in the server database.

A variation is represented as an object with the following fields:

uri (*uri*) URI for this resource.

task (*object*) Task information, see *Tasked resources*.

data_source (*object*) *Link* to a *data source* resource (embeddable).

sample (*object*) *Link* to a *sample* resource (embeddable).

Collection

See also:

Collection resources

`GET /variations/`
Returns a collection of variations in the *variation_collection* field.

Note: Requires one or more of the following:

- Having the *admin* role.
- Being the owner of the sample specified by the *sample* filter.
- Setting the *sample* filter to a public sample.

Available filters:

- **sample** (*uri*)

POST /variations/
Adds a variation resource.

Note: Requires having the *admin* role or being the owner of the sample specified by the *sample* field.

Required request data:

- **data_source** (*uri*)
- **sample** (*uri*)

Instances

GET /variations/<id>
Returns the variation representation in the *variation* field.

Note: Requires having the *admin* role or being the owner of the variation.

PATCH /variations/<id>
Updates a variation resource.

Note: Requires having the *admin* role.

Accepted request data:

- **task** (*object*)

CHAPTER 5

Additional notes

This part contains some notes for developers and other random notes. It needs work, sorry about that.

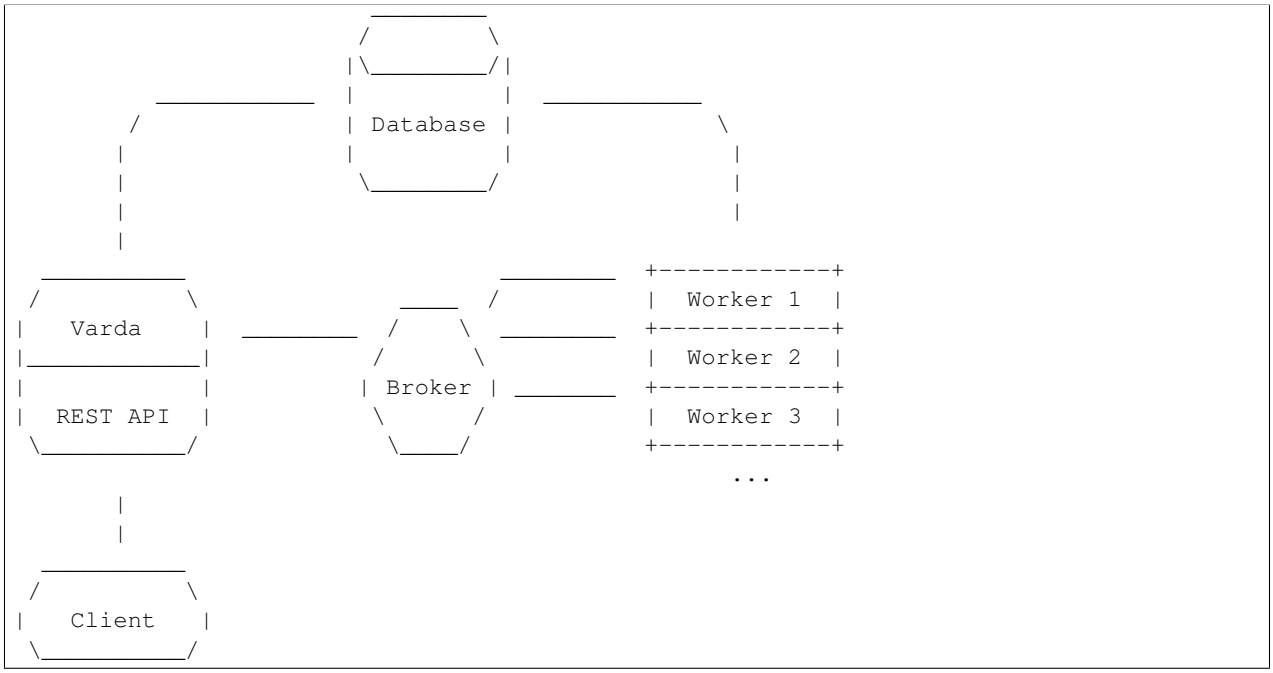
5.1 Application design

Note: Work in progress, some of this should probably be refactored into user documentation.

5.1.1 Implementation

Varda is implemented on top of the Flask web microframework, the Celery distributed task queue, and the SQLAlchemy object relational mapper.

A typical deployment looks like this:



5.1.2 Sample types

Three types of samples:

1. Simple sample (one individual)
2. Pooled sample (multiple individuals)
3. Population study

The following table gives a summary of what is stored for each sample type:

Stored data	Simple sample	Pooled sample	Population study
Pool size	1	N	N
Variant observations	O with support 1	N x O with support 1	O with support N
Covered regions	R	N x R	None

- N = number of individuals
- O = number of observations per individual
- R = number of covered regions per individual

For all sample types, data can be imported from an arbitrary number of data sources. This means you could for example import variant observations per individual, per chromosome, or per variant type.

Pooled samples can have their individuals effectively anonymized by importing variant observations from one big data source in which the order is not related to the individuals. For example, `cat` the VCF files for all individuals and sort the result by genome position before sending it to the server.

5.1.3 Frequency calculation

Only take samples into account with covered regions (to rule out population studies).

5.1.4 Binning of regions and observations

Todo. UCSC binning scheme.

5.1.5 Security

Todo: Add a page on security to the Managing Varda section.

Authentication via HTTP Basic Authentication, or API tokens, so only use with SSL.

5.1.6 Sample state

A sample can be either *active* or *inactive* (default). An inactive sample is ignored in any frequency calculations.

Importing data sources is only possible for inactive samples. A sample cannot be made active while any data source is being imported for that sample. Users can only make a sample active, not inactive.

5.2 Todo list

These are some general todo notes. More specific notes can be found by grepping the source code for `Todo`.

- More strict validation of user input, especially file uploads (max file size and contents).
- Implement caching control headers.
- Implement HEAD requests.
- Better organised and more comprehensive test suite.
- Throtling.
- Better rights/roles model.
- Support input in BCF2 format.
- Have a look at supporting the gVCF format.
- Possibility to contact submitter of an observation.
- Have a maintenance and/or read-only mode, probably with HTTP redirects.
- Store phasing info, for example by numbering each allele (uniquely within a sample) and store the allele number with observations.
- Support bigBed format.
- What to do for variants where we have more observations than coverage? We could have a check in sample activation, but would we really like to enforce this?
- Fallback modes to accomodate browsing the API with a standard web browser, e.g., query string alternative to pagination with Accept-Range headers. Perhaps this can be optional and implemented by patching the Request object before it reaches the API code.
- We currently store variants as *(position, reference, observed)* and regions as *(begin, end)* where all positioning is one-based and inclusive. An alternative is implemented in the `observation-format` git branch where all positioning is zero-based and open-ended and variants are stored as *(begin, end, observed)*.

  Here are some advantages of the alternative representation:

  - If a reference genome is configured, the *reference* field is superfluous and we can do with defining just a region.
  - Zero-based and open-ended positioning follows Python indexing and slicing notation as well as the BED format.
  - Insertions are perhaps more naturally modelled by giving an empty region on the reference genome.
  - Overlaps between regions and variants are easier to query for with *begin* and *end* fields.

  But it also has some downsides:

  - The current variant representation follows existing practices and therefore all interfaces to the outside world more closely.
  - If there is no reference genome configured, we don't have a complete definition of our variants.
  - It means a lot of conversions between representations.

  Note that the current representation isn't following VCF, since VCF requires both the *reference* and *observed* sequences to be non-empty. However, by normalizing (and also anticipating other sources than VCF) we trim every sequence as much as possible.

  For now we think it is best to stick with the current representations, but this is still somewhat up for discussion.

- Have a section in the docs describing the unit tests. Also note that the unit tests use the first 200,000 bases of chromosome 19 as a reference genome.
- Refactor how we handle Celery tasks. Don't store the task uuid in the database. Probably also create the resulting resource in the task, not before starting the task like we do now.

  A running task should be monitored and, when finished, it points to the resulting resource.

  We can probably still list running tasks even though we don't store them in the database, following what Flower does. This will only work when sending task events is enabled (`-E` option to `celeryd`). Also have a look at *CELERY_SEND_EVENTS* and *CELERY_SEND_TASK_SENT_EVENT* configuration options. As this post suggests, we probably also have to explicitely monitor the events.

  Important: We still seem to have an issue with many long-running tasks where some of them may be run twice. In general, this will raise the `TaskError('variation_imported', 'Variation already imported')` exception but I have seen at least one case where the entire variation has been imported twice which is quite hard to recover from. My hope is that we can prevent this from happening by some refactoring here.
- See if this issue affects us.
- For simplicity, we are currently storing *homozygous* vs *heterozygous* for each alternate call. Shouldn't we actually be storing the genotype, like *0/1* vs *1/1* (in reporting, we could include *0/0*)? It is more general.

  I can think of two reasons why we choose not to store genotypes. The first is that we don't have reference calls (but we could simply omit *0/0*). The second is that we don't have a guarantee that a given chromosome was called using the same ploidity. Therefore, we could for example have genotypes from different samples on the Y chromosome as *0/0*, *0/1*, *1/1* versus *0*, *1*. We could report these as-is, or merge them to the highest ploidity which would be incorrect in this case. Or we store the ploidity for each chromosome system-wide.
- Having a pool size per sample is not granular enough in some situations. For example, the 1KG phase1 integrated call sets are over 1092 individuals for most chromosomes, but over 1083 and 535 for the mitochondrial genome and chromosome Y, respectively. Not sure if we can really solve this easily, since having a pool size per variation/coverage will not work for samples with coverage.
- Options for logging in a production environment. Basically, if *DEBUG=False*, everything from log level warning and up should be logged to a file and every error should optionally be e-mailed.
- JSON is not a hypertext format, but still we can do better by using hypertext-like representations, for example using HAL.
- Replace *Resource* base class by *SingletonResource* and *CollectionResource*. Implement the root, genome, and authentication resource using *SingletonResource*.
- See if we can easily compress with bgzip instead of regular gzip.
- Perhaps use Factory Boy instead of fixture. It looks like we don't have to monkey patch Factory Boy.
- Use JSONPatch for editing resources (example).

5.3 Changelog

Here you can see the full list of changes between each Varda release.

5.3.1 Version 0.1.0

Release date to be decided.

First public release.

5.4 Copyright

Varda is licensed under the MIT License, meaning you can do whatever you want with it as long as all copies include these license terms. The full license text can be found below.

The profile picture for the Varda GitHub organisation was cropped from an artist's rendition of Varda Elentári, Queen of the Stars by Dominik Matus and is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

5.4.1 Authors

Varda is written and maintained by Martijn Vermaat, based on a prototype by Jeroen Laros.

- Leiden University Medical Center <humgen@lumc.nl>
- Martijn Vermaat <martijn@vermaat.name>
- Jeroen Laros <j.f.j.laros@vermaat.name>

5.4.2 License

5.5 Links

Some links to hopefully relevant resources on the web.

- Modern Web Application Architecture
- Getting RESTful with web.py
- Simple Flask extension for RESTful APIs
- Example: Glpi Rest Api
- Example: The datahub
- Example: Andalucia
- A skeleton for Flask applications
- SQLAlchemy usage recipes

- Dropbox API
- Twitter API
- Amazon S3 REST API
- Designing a Secure REST (Web) API without OAuth
- bitly API Documentation
- Alembic Documentation
- What exactly is RESTful programming?
- General principles for good URI design for RESTful and HTTP applications
- How I learned to stop worrying and love REST
- REST Done Right with Steve Klabnik podcast
- Django REST framework
- Designing a RESTful Web API
- Hypermedia APIs - Jon Moore
- Normalized variant representation
- Correctly use HTTP verbs
- Thoughts on RESTful API Design
- Developing RESTful Web APIs with Python, Flask and MongoDB
- Design Beautiful REST + JSON APIs
- GenomeSpace: RESTful Access to Data Manager
- Designing a Pragmatic RESTful API
- Best Practices for Designing a Pragmatic RESTful API

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

/

/annotations

/coverages

/data_sources

/groups

/samples

/tokens

/users

/variants

/variations