Global Alliance
for Genomics & Health

# GA4GH Documentation

***Release 0.2.0a2.dev19+ng5118fa7***

**Global Alliance for Genomics and Health**

January 14, 2016

Contents

Global Alliance
for Genomics & Health

This the documentation for version 0.2.0a2.dev19 of the GA4GH reference implementation.

CHAPTER 1

# Contents

## 1.1 Introduction

The Data Working Group of the Global Alliance for Genomics and Health has defined an API to facilitate interoperable exchange of genomic data. This is the the documentation for the reference implementation of the API.

**Simplicity/clarity** The main goal of this implementation is to provide an easy to understand and maintain implementation of the GA4GH API. Design choices are driven by the goal of making the code as easy to understand as possible, with performance being of secondary importance. With that being said, it should be possible to provide a functional implementation that is useful in many cases where the extremes of scale are not important.

**Portability** The code is written in Python for maximum portability, and it should be possible to run on any modern computer/operating system (Windows compatibility should be possible, although this has not been tested). Our coding guidelines specify using a subset of Python 3 which is backwards compatible with Python 2 following the current best practices. The project currently does not yet support Python 3, as support for it is lacking in several packages that we depend on. However, our eventual goal is to support both Python 2 and 3.

**Ease of use** The code follows the Python Packaging User Guide. Specifically, pip is used to handle python package dependencies (see below for details). This provides easy installation of the `ga4gh` reference code across a range of operating systems.

## 1.2 GA4GH API Demo

In this demo, we'll install a copy of the GA4GH reference implementation and run a local version of the server using some example data. We then run some example queries on this server using various different methods to illustrate the basics of the protocol. The server can, of course, be run on any machine on the network, but for simplicity we assume that the client and the server are running on your local machine during this demo.

The instructions for installation here are not intended to be used in a production deployment. See the *Installation* section for a detailed guide on production installation. To run the demo, you will need a working installation of Python 2.7 and also have virtualenv installed. We also need to have zlib and a few other common libraries installed so that we can build some of the packages that the reference server depends on.

On Debian/Ubuntu, for example, we can install these packages using:

```
$ sudo apt-get install python-dev python-virtualenv zlib1g-dev libxslt1-dev libffi-dev libssl-dev
```

On Fedora 22+ (current), the equivalent would be:

```
$ sudo dnf install python-devel python-virtualenv zlib-devel libxslt-devel openssl-devel
```

First, we create a virtualenv sandbox to isolate the demo from the rest of the system, and then activate it:

```
$ virtualenv ga4gh-env
$ source ga4gh-env/bin/activate
```

Now, install the ga4gh package from the Python package index. This will take some time, as some upstream packages will need to be built and installed.

```
(ga4gh-env) $ pip install ga4gh --pre
```

(Older versions of pip might not recognise the `--pre` argument; if not, it is safe to remove it.)

Now we can download some example data, which we'll use for our demo:

```
(ga4gh-env) $ wget http://www.well.ox.ac.uk/~jk/ga4gh-example-data-v3.0.tar
(ga4gh-env) $ tar -xvf ga4gh-example-data-v3.0.tar
```

After extracting the data, we can then run the `ga4gh_server` application:

```
(ga4gh-env) $ ga4gh_server
* Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
* Restarting with stat
```

(The server is using a default configuration which assumes the existence of the `ga4gh-example-data` directory for simplicity here; see the *Configuration* section for detailed information on how we configure the server.) We now have a server running in the foreground. When it receives requests, it will print out log entries to the terminal. A summary of the server's configuration and data is available in HTML format at `http://locahost:8000`, which can be viewed in a web browser. Leave the server running and open another terminal to complete the rest of the demo.

To try out the server, we must send some requests to it using the GA4GH protocol. One way in which we can do this is to manually create the JSON requests, and send these to the server using cURL:

```
$ curl --data '{}' --header 'Content-Type: application/json' \
http://localhost:8000/datasets/search | jq .
```

In this example, we used the searchDatasets method to ask the server for all the Datasets on the server. It responded by sending back some JSON, which we piped into the jq JSON processor to make it easier to read. We get the following result:


```
{
  "nextPageToken": null,
  "datasets": [
    {
      "description": null,
      "name": "1kg-p3-subset",
      "id": "MWtnLXAzLXN1YnNldA=="
    }
  ]
}
```


In this example we sent a SearchDatasetsRequest object to the server and received a SearchDatasetsResponse object in return. This response object contained one Dataset object, which is contained in the `datasets` array. This approach to interacting with the server is tedious and error prone, as we have to hand-craft the request objects. It is also quite inconvenient, as we may have to request many pages of objects to get all the objects that satisfy our search criteria.

To simplify interacting with the server and to abstract away the low-level network-level details of the server, we provide a client application. To try this out, we start another instance of our virtualenv, and then send the equivalent command using:

```
$ source ga4gh-env/bin/activate
(ga4gh-env) $ ga4gh_client datasets-search http://localhost:8000
```

```
MWtnLXAzLXN1YnNldA==    1kg-p3-subset
```

The output of this command is a summary of the Datasets on that are present on the server. We can also get the output in JSON form such that each object is written on one line:

```
(ga4gh-env) $ ga4gh_client datasets-search -O json http://localhost:8000
```

```
{"description": null, "name": "1kg-p3-subset", "id": "MWtnLXAzLXN1YnNldA=="}
```

This format is quite useful for larger queries, and can be piped into jq to extract fields of interest, pretty printing and so on.

We can perform similar queries for variant data using the searchVariants API call. First, we find the IDs of the VariantSets on the server using the searchVariantSets method:

```
(ga4gh-env) $ ga4gh_client variantsets-search http://localhost:8000
```

```
MWtnLXAzLXN1YnNldDptdm5jYWxs    mvncall
```

This tells us that we have one VariantSet on the server, with ID `MWtnLXAzLXN1YnNldDptdm5jYWxs` and name `mvncall`. We can then search for variants overlapping a given interval in a VariantSet as follows:

```
(ga4gh-env) $ ga4gh_client variants-search http://localhost:8000 \
--referenceName=1 --start=45000 --end=50000
```

The output of the client program is a summary of the data received in a free text form. This is not intended to be used as the input to other programs, and is simply a data exploration tool for users. To really *use* our data, we should use a GA4GH client library.

Part of the GA4GH reference implementation is a *Python Client Library*. This makes sending requests to the server and using the responses very easy. For example, to run the same query as we performed above, we can use the following code:

```
from __future__ import print_function

import ga4gh.client as client

httpClient = client.HttpClient("http://localhost:8000")
# Get the datasets on the server.
datasets = list(httpClient.searchDatasets())
# Get the variantSets in the first dataset.
variantSets = list(httpClient.searchVariantSets(
    datasetId=datasets[0].id))
# Now get the variants in the interval [45000, 50000) on chromosome 1
# in the first variantSet.
iterator = httpClient.searchVariants(
    variantSetId=variantSets[0].id,
    referenceName="1", start=45000, end=50000)
for variant in iterator:
    print(
        variant.referenceName, variant.start, variant.end,
        variant.referenceBases, variant.alternateBases, sep="\t")
```

If we save this script as `ga4gh-demo.py` we can then run it using:

```
(ga4gh-env) $ python ga4gh-demo.py
```

**Todo**

Add more examples of using the reads API and give examples of using the references API. We should aim to have a single complete example, where we start with a given variant, and drill down into the reads in question programatically. values as parameters which have sensible defaults.

### 1.2.1 With OIDC

**Todo**

Should we move the OIDC documentation into its own section? It is quite a lot of complication to add here to a beginners HOWTO.

If we want authentication, we must have an OIDC authentication provider. One can be found in `oidc-provider`, and run with the `run.sh` script. We can then use this with the `LocalOidConfig` server configuration. So:

```
$ cd oidc-provider && ./run.sh
```

In another shell on the same machine

```
$ python server_dev.py -c LocalOidConfig
```

Make sure you know the hostname the server is running on. It can be found with

```
$ python -c 'import socket; print socket.gethostname()'
```

With a web browser, go to `https://<server hostname>:<server port>`. You may need to accept the security warnings as there are probably self-signed certificates. You will be taken through an authentication flow. When asked for a username and password, try `upper` and `crust`. You will find yourself back at the ga4gh server homepage. On the homepage will be a 'session token' This is the key to access the server with the client tool as follows:

```
(ga4gh-env) $ ga4gh_client --key <key from homepage> variantsets-search https://localhost:8000/curre
MWtnLXAzLXN1YnNldDptdm5jYWxs    mvncall
```

## 1.3 Installation

This section documents the process of deploying the GA4GH reference server in a production setting. The intended audience is therefore server administrators. If you are looking for a quick demo of the GA4GH API using a local installation of the reference server please check out the *GA4GH API Demo*. If you are looking for instructions to get a development system up and running, then please go to the *Development* section.

### 1.3.1 Deployment on Apache

To deploy on Apache on Debian/Ubuntu platforms, do the following.

First, we install some basic pre-requisite packages:

```
$ sudo apt-get install python-dev python-virtualenv zlib1g-dev libxslt1-dev libffi-dev libssl-dev
```

Install Apache and mod_wsgi, and enable mod_wsgi:

```
$ sudo apt-get install apache2 libapache2-mod-wsgi
$ sudo a2enmod wsgi
```

Create the Python egg cache directory, and make it writable by the `www-data` user:

```
$ sudo mkdir /var/cache/apache2/python-egg-cache
$ sudo chown www-data:www-data /var/cache/apache2/python-egg-cache/
```

Create a directory to hold the GA4GH server code, configuration and data. For convenience, we make this owned by the current user (but make sure all the files are world-readable).:

```
$ sudo mkdir /srv/ga4gh
$ sudo chown $USER /srv/ga4gh
$ cd /srv/ga4gh
```

Make a virtualenv, and install the ga4gh package:

```
$ virtualenv ga4gh-server-env
$ source ga4gh-server-env/bin/activate
(ga4gh-server-env) $ pip install --pre ga4gh  # We need the --pre because ga4gh is pre-release
(ga4gh-server-env) $ deactivate
```

Download and unpack the example data:

```
$ wget http://www.well.ox.ac.uk/~jk/ga4gh-example-data-v3.0.tar
$ tar -xf ga4gh-example-data-v3.0.tar
```

Create the WSGI file at `/srv/ga4gh/application.wsgi` and write the following contents:

```
from ga4gh.frontend import app as application
import ga4gh.frontend as frontend
frontend.configure("/srv/ga4gh/config.py")
```

Create the configuration file at `/srv/ga4gh/config.py`, and write the following contents:

```
DATA_SOURCE = "/srv/ga4gh/ga4gh-example-data"
```

(Many more configuration options are available — see the *Configuration* section for a detailed discussion on the server configuration and input data.)

Configure Apache. Edit the file `/etc/apache2/sites-enabled/000-default.conf` and insert the following contents towards the end of the file (*within* the `<VirtualHost:80>...</VirtualHost>` block):

```
WSGIDaemonProcess ga4gh \
    python-path=/srv/ga4gh/ga4gh-server-env/lib/python2.7/site-packages \
    python-eggs=/var/cache/apache2/python-egg-cache
WSGIScriptAlias /ga4gh /srv/ga4gh/application.wsgi

<Directory /srv/ga4gh>
    WSGIProcessGroup ga4gh
    WSGIApplicationGroup %{GLOBAL}
    Require all granted
</Directory>
```

Restart Apache:

```
$ sudo service apache2 restart
```

Test the installation by pointing a web-browser at the root URL; for example, to test on the installation server use:

```
$ links http://localhost/ga4gh
```

We can also test the server by running some API commands; the instructions in the *GA4GH API Demo* can be easily adapted here to test out the server across the network.

There are any number of different ways in which we can set up a WSGI application under Apache, which may be preferable in different installations. (In particular, the Apache configuration here may be specific to Ubuntu 14.04, where this was tested.) See the mod_wsgi documentation for more details. These instructions are also specific to Debian/Ubuntu and different commands and directory structures will be required on different platforms.

The server can be deployed on any WSGI compliant web server. See the instructions in the Flask documentation for more details on how to deploy on various other servers.

**TODO**

1. Add more detail on how we can test out the API by making some client queries.
2. Add links to the Configuration section to give details on how we configure the server.

#### Troubleshooting

If you are encountering difficulties getting the above to work, it is helpful to turn on debugging output. Do this by adding the following line to your config file:

```
DEBUG = True
```

When an error occurs, the details of this will then be printed to the web server's error log (in Apache on Debian/Ubuntu, for example, this is `/var/log/apache2/error.log`).

### 1.3.2 Deployment on Docker

It is also possible to deploy the server using Docker.

First, you need an environment running the docker daemon. For non-production use, we recommend boot2docker. For production use you should install docker on a stable linux distro. Please reference the platform specific Docker installation instructions. OSX and Windows are instructions for boot2docker.

**Local Dataset Mounted as Volume**

If you already have a dataset on your machine, you can download and deploy the apache server in one command:

```
$ docker run -e GA4GH_DATA_SOURCE=/data -v /my/ga4gh_data/:/data:ro -d -p 8000:80 --name ga4gh_server
```

Replace `/my/ga4gh_data/` with the path to your data.

This will:

- pull the automatically built image from Dockerhub
- start an apache server running mod_wsgi on container port 80
- mount your data read-only to the docker container
- assign a name to the container
- forward port 8000 to the container.

For more information on docker run options, see the run reference.

**Demo Dataset Inside Container**

If you do not have a dataset yet, you can deploy a container which includes the demo data:

```
$ docker run -d -p 8000:80 --name ga4gh_demo afirth/ga4gh-server:develop-demo
```

This is identical to the production container, except that a copy of the demo data is included and appropriate defaults are set.

**Developing Client Code: Run a Client Container and a Server**

In this example you run a server as a daemon in one container, and the client as an ephemeral instance in another container. From the client, the server is accessible at `http://server/`, and the `/tmp/mydev` directory is mounted at `/app/mydev/`. Any changes you make to scripts in `mydev` will be reflected on the host and container and persist even after the container dies.

```
#make a development dir and place the example client script in it
$ mkdir /tmp/mydev
$ curl https://raw.githubusercontent.com/ga4gh/server/develop/scripts/demo_example.py > /tmp/mydev/de
$ chmod +x /tmp/mydev/demo_example.py

# start the server daemon
# assumes the demo data on host at /my/ga4gh_data
$ docker run -e GA4GH_DEBUG=True -e GA4GH_DATA_SOURCE=/data -v /my/ga4gh_data/:/data:ro -d --name ga4

# start the client and drop into a bash shell, with mydev/ mounted read/write
# --link adds a host entry for server, and --rm destroys the container when you exit
$ docker run -e GA4GH_DEBUG=True -v /tmp/mydev/:/app/mydev:rw -it --name ga4gh_client --link ga4gh_se

# call the client code script
root@md5:/app# ./mydev/demo_example.py

# call the command line client
root@md5:/app# ga4gh_client variantsets-search http://server/current

#exit and destroy the client container
root@md5:/app# exit
```

**Ports**

The `-p 8000:80` argument to `docker run` will run the docker container in the background, and translate calls from your host environment port 8000 to the docker container port 80. At that point you should be able to access it like a normal website, albeit on port 8000. Running in boot2docker, you will need to forward the port from the boot2docker VM to the host. From a terminal on the host to forward traffic from localhost:8000 to the VM 8000 on OSX:

```
$ VBoxManage controlvm boot2docker-vm natpf1 "ga4gh,tcp,127.0.0.1,8000,,8000"
```

For more info on port forwarding see the VirtualBox manual and this wiki article.

### Advanced

If you want to build the images yourself, that is possible. The afirth/ga4gh-server repo builds automatically on new commits, so this is only needed if you want to modify the Dockerfiles, or build from a different source.

The prod and demo builds are based off of mod_wsgi-docker, a project from the author of mod_wsgi. Please reference the Dockerfiles and documentation for that project during development on these builds.

**Examples**

Build the code at server/ and run for production, serving a dataset on local host located at `/my/dataset`

```
$ cd server/
$ docker build -t my-repo/my-image .
$ docker run -e GA4GH_DATA_SOURCE=/dataset -v /my/dataset:/dataset:ro -itd -p 8000:80 --name ga4gh_se
```

Build and run the production build from above, with the demo dataset in the container (you will need to modify the FROM line in `/deploy/variants/demo/Dockerfile` if you want to use your image from above as the base):

```
$ cd server/deploy/variants/demo
$ docker build -t my-repo/my-demo-image .
$ docker run -itd -p 8000:80 --name ga4gh_demo my-repo/my-demo-image
```

**Variants**

Other Dockerfile implementations are available in the variants folder which install manually. To build one of these images:

```
$ cd server/deploy/variants/xxxx
$ docker build -t my-repo/my-image .
$ docker run -itd -p 8000:80 --name my_container my-repo/my-image
```

### Troubleshooting Docker

**DNS**

The docker daemon's DNS may be corrupted if you switch networks, especially if run in a VM. For boot2docker, running udhcpc on the VM usually fixes it. From a terminal on the host:

```
$ eval "$(boot2docker shellinit)"
$ boot2docker ssh
>     sudo udhcpc
(password is tcuser)
```

**DEBUG**

To enable DEBUG on your docker server, call docker run with `-e GA4GH_DEBUG=True`

```
$ docker run -itd -p 8000:80 --name ga4gh_demo -e GA4GH_DEBUG=True afirth/ga4gh-server:develop-demo
```

This will set the environment variable which is read by config.py

You can then get logs from the docker container by running `docker logs (container)` e.g. `docker logs ga4gh_demo`

## 1.4 Python Client Library

This is the GA4GH client API library. This is a convenient wrapper for the low-level HTTP GA4GH API, and abstracts away network centric details such as paging. The methods and types used by the client library are defined by the GA4GH schema.

> **Warning:** This client API should be considered early alpha quality, and may change in arbitrary ways. In particular, the current camelCase convention for identifiers may change to snake_case in future.

**Todo**

A full description of this API and links to a tutorial on how to use it, as well as a quickstart showing the basic usage.

### 1.4.1 Types

**Todo**

A short overview of the types and links to the high-level docs.

#### References

**class** ga4gh.protocol.**ReferenceSet**(***kwargs*)

A ReferenceSet is a set of References which typically comprise a reference assembly, such as GRCh38. A ReferenceSet defines a common coordinate space for comparing reference-aligned experimental data.

**assemblyId**
Public id of this reference set, such as GRCh37.

**description**
Optional free text description of this reference set.

**id**
The reference set ID. Unique in the repository.

**isDerived**
A reference set may be derived from a source if it contains additional sequences, or some of the sequences within it are derived (see the definition of isDerived in Reference).

**md5checksum**
Order-independent MD5 checksum which identifies this ReferenceSet. To compute this checksum, make a list of Reference.md5checksum for all References in this set. Then sort that list, and take the MD5 hash of all the strings concatenated together. Express the hash as a lower-case hexadecimal string.

**name**
The reference set name.

**ncbiTaxonId**
ID from http://www.ncbi.nlm.nih.gov/taxonomy (e.g. 9606->human) indicating the species which this assembly is intended to model. Note that contained References may specify a different ncbiTaxonId, as assemblies may contain reference sequences which do not belong to the modeled species, e.g. EBV in a human reference genome.

**sourceAccessions**
All known corresponding accession IDs in INSDC (GenBank/ENA/DDBJ) ideally with a version number, e.g. NC_000001.11.

**sourceURI**
Specifies a FASTA format file/string.

**class** ga4gh.protocol.**Reference**(***kwargs*)

A Reference is a canonical assembled contig, intended to act as a reference coordinate space for other genomic annotations. A single Reference might represent the human chromosome 1, for instance. References are designed to be immutable.

**id**
The reference ID. Unique within the repository.

**isDerived**
A sequence X is said to be derived from source sequence Y, if X and Y are of the same length and the per-base sequence divergence at A/C/G/T bases is sufficiently small. Two sequences derived from the same official sequence share the same coordinates and annotations, and can be replaced with the official sequence for certain use cases.

**length**
The length of this reference's sequence.

**md5checksum**
The MD5 checksum uniquely representing this Reference as a lower-case hexadecimal string, calculated as the MD5 of the upper-case sequence excluding all whitespace characters (this is equivalent to SQ:M5 in SAM).

**name**
The name of this reference. (e.g. '22').

**ncbiTaxonId**
ID from http://www.ncbi.nlm.nih.gov/taxonomy (e.g. 9606->human).

**sourceAccessions**
All known corresponding accession IDs in INSDC (GenBank/ENA/DDBJ) which must include a version number, e.g. GCF_000001405.26.

**sourceDivergence**
The sourceDivergence is the fraction of non-indel bases that do not match the reference this record was derived from.

**sourceURI**
The URI from which the sequence was obtained. Specifies a FASTA format file/string with one name, sequence pair. In most cases, clients should call the getReferenceBases() method to obtain sequence bases for a Reference instead of attempting to retrieve this URI.

### Datasets

**class** ga4gh.protocol.**Dataset**(***kwargs*)
A Dataset is a collection of related data of multiple types. Data providers decide how to group data into datasets. See [Metadata API](../api/metadata.html) for a more detailed discussion.

**description**
Additional, human-readable information on the dataset.

**id**
The dataset's id, locally unique to the server instance.

**name**
The name of the dataset.

### Variants

**class** ga4gh.protocol.**VariantSet**(***kwargs*)
A VariantSet is a collection of variants and variant calls intended to be analyzed together.

**datasetId**
The ID of the dataset this variant set belongs to.

**id**
The variant set ID.

**metadata**
Optional metadata associated with this variant set. This array can be used to store information about the variant set, such as information found in VCF header fields, that isn't already available in first class fields such as "name".

**name**
The variant set name.

**referenceSetId**
The ID of the reference set that describes the sequences used by the variants in this set.

**class** ga4gh.protocol.**CallSet**(***kwargs*)
A CallSet is a collection of calls that were generated by the same analysis of the same sample.

**created**
The date this call set was created in milliseconds from the epoch.

**id**
The call set ID.

**info**
A map of additional call set information.

**name**
The call set name.

**sampleId**
The sample this call set's data was generated from. Note: the current API does not have a rigorous definition of sample. Therefore, this field actually contains an arbitrary string, typically corresponding to the sampleId field in the read groups used to generate this call set.

**updated**
The time at which this call set was last updated in milliseconds from the epoch.

**variantSetIds**
The IDs of the variant sets this call set has calls in.

**class** ga4gh.protocol.**Variant**(***kwargs*)
A Variant represents a change in DNA sequence relative to some reference. For example, a variant could represent a SNP or an insertion. Variants belong to a VariantSet. This is equivalent to a row in VCF.

**alternateBases**
The bases that appear instead of the reference bases. Multiple alternate alleles are possible.

**calls**
The variant calls for this particular variant. Each one represents the determination of genotype with respect to this variant. Calls in this array are implicitly associated with this Variant.

**created**
The date this variant was created in milliseconds from the epoch.

**end**
The end position (exclusive), resulting in [start, end) closed-open interval. This is typically calculated by start + referenceBases.length.

**id**
The variant ID.

**info**
A map of additional variant information.

**names**
Names for the variant, for example a RefSNP ID.

**referenceBases**
The reference bases for this variant. They start at the given start position.

**referenceName**
The reference on which this variant occurs. (e.g. chr20 or X)

**start**
The start position at which this variant occurs (0-based). This corresponds to the first base of the string of reference bases. Genomic positions are non-negative integers less than reference length. Variants spanning the join of circular genomes are represented as two variants one on each side of the join (position 0).

**updated**
The time at which this variant was last updated in milliseconds from the epoch.

**variantSetId**
The ID of the VariantSet this variant belongs to. This transitively defines the ReferenceSet against which the Variant is to be interpreted.

### Reads

**class** ga4gh.protocol.**ReadGroupSet**(***kwargs*)
A ReadGroupSet is a logical collection of ReadGroups. Typically one ReadGroupSet represents all the reads from one experimental sample.

**datasetId**
The ID of the dataset this read group set belongs to.

**id**
The read group set ID.

**name**
The read group set name.

**readGroups**
The read groups in this set.

**stats**
Statistical data on reads in this read group set.

**class** ga4gh.protocol.**ReadGroup**(***kwargs*)
A ReadGroup is a set of reads derived from one physical sequencing process.

**created**
The time at which this read group was created in milliseconds from the epoch.

**datasetId**
The ID of the dataset this read group belongs to.

**description**
The read group description.

**experiment**
The experiment used to generate this read group.

**id**
The read group ID.

**info**
A map of additional read group information.

**name**
The read group name.

**predictedInsertSize**
The predicted insert size of this read group.

**programs**
The programs used to generate this read group.

**referenceSetId**
The ID of the reference set to which the reads in this read group are aligned. Required if there are any read alignments.

**sampleId**
The sample this read group's data was generated from. Note: the current API does not have a rigorous definition of sample. Therefore, this field actually contains an arbitrary string, typically corresponding to the SM tag in a BAM file.

**stats**
Statistical data on reads in this read group.

**updated**
The time at which this read group was last updated in milliseconds from the epoch.

**class** ga4gh.protocol.**ReadAlignment**(***kwargs*)
Each read alignment describes an alignment with additional information about the fragment and the read. A read alignment object is equivalent to a line in a SAM file.

**alignedQuality**
The quality of the read sequence contained in this alignment record (equivalent to QUAL in SAM). alignedSequence and alignedQuality may be shorter than the full read sequence and quality. This will occur if the alignment is part of a chimeric alignment, or if the read was trimmed. When this occurs, the CIGAR for this read will begin/end with a hard clip operator that will indicate the length of the excised sequence.

**alignedSequence**
The bases of the read sequence contained in this alignment record (equivalent to SEQ in SAM). alignedSequence and alignedQuality may be shorter than the full read sequence and quality. This will occur if the alignment is part of a chimeric alignment, or if the read was trimmed. When this occurs, the CIGAR for this read will begin/end with a hard clip operator that will indicate the length of the excised sequence.

**alignment**
The alignment for this alignment record. This field will be null if the read is unmapped.

**duplicateFragment**
The fragment is a PCR or optical duplicate (SAM flag 0x400).

**failedVendorQualityChecks**
The read fails platform or vendor quality checks (SAM flag 0x200).

**fragmentLength**
The observed length of the fragment, equivalent to TLEN in SAM.

**fragmentName**
The fragment name. Equivalent to QNAME (query template name) in SAM.

**id**
The read alignment ID. This ID is unique within the read group this alignment belongs to. For performance reasons, this field may be omitted by a backend. If provided, its intended use is to make caching and UI display easier for genome browsers and other lightweight clients.

**info**
A map of additional read alignment information.

**nextMatePosition**
The mapping of the primary alignment of the (readNumber+1)%numberReads read in the fragment. It replaces mate position and mate strand in SAM.

**numberReads**
The number of reads in the fragment (extension to SAM flag 0x1)

**properPlacement**
The orientation and the distance between reads from the fragment are consistent with the sequencing protocol (equivalent to SAM flag 0x2)

**readGroupId**
The ID of the read group this read belongs to. (Every read must belong to exactly one read group.)

**readNumber**
The read ordinal in the fragment, 0-based and less than numberReads. This field replaces SAM flag 0x40 and 0x80 and is intended to more cleanly represent multiple reads per fragment.

**secondaryAlignment**
Whether this alignment is secondary. Equivalent to SAM flag 0x100. A secondary alignment represents an alternative to the primary alignment for this read. Aligners may return secondary alignments if a read can map ambiguously to multiple coordinates in the genome. By convention, each read has one and only one alignment where both secondaryAlignment and supplementaryAlignment are false.

**supplementaryAlignment**
Whether this alignment is supplementary. Equivalent to SAM flag 0x800. Supplementary alignments are used in the representation of a chimeric alignment. In a chimeric alignment, a read is split into multiple linear alignments that map to different reference contigs. The first linear alignment in the read will be designated as the representative alignment; the remaining linear alignments will be designated as supplementary alignments. These alignments may have different mapping quality scores. In each linear alignment in a chimeric alignment, the read will be hard clipped. The alignedSequence and alignedQuality fields in the alignment record will only represent the bases for its respective linear alignment.

**class** ga4gh.protocol.**Position**(***kwargs*)
A Position is an unoriented base in some Reference. A Position is represented by a Reference name, and a base number on that Reference (0-based).

**position**
The 0-based offset from the start of the forward strand for that Reference. Genomic positions are non-negative integers less than Reference length.

**referenceName**
The name of the Reference on which the Position is located.

**strand**
Strand the position is associated with.

### 1.4.2 Client API

**Todo**

Add overview documentation for the client API.

**class** ga4gh.client.**HttpClient**(*urlPrefix*, *logLevel=30*, *authenticationKey=None*)
The GA4GH HTTP client. This class provides methods corresponding to the GA4GH search and object GET methods.

**Todo**

Add a better description of the role of this class and include links to the high-level API documention.

**Parameters**

- **urlPrefix** (*str*) – The base URL of the GA4GH server we wish to communicate with. This should include the 'http' or 'https' prefix.
- **logLevel** (*int*) – The amount of debugging information to log using the `logging` module. This is `logging.WARNING` by default.
- **authenticationKey** (*str*) – The authentication key provided by the server after logging in.

**getDataset** (*datasetId*)
Returns the Dataset with the specified ID from the server.

**Parameters** **datasetId** (*str*) – The ID of the Dataset of interest.

**Returns** The Dataset of interest.

**Return type** *ga4gh.protocol.Dataset*

**getReadGroup** (*readGroupId*)
Returns the ReadGroup with the specified ID from the server.

**Parameters** **readGroupId** (*str*) – The ID of the ReadGroup of interest.

**Returns** The ReadGroup of interest.

**Return type** *ga4gh.protocol.ReadGroup*

**getReadGroupSet** (*readGroupSetId*)
Returns the ReadGroupSet with the specified ID from the server.

**Parameters** **readGroupSetId** (*str*) – The ID of the ReadGroupSet of interest.

**Returns** The ReadGroupSet of interest.

**Return type** *ga4gh.protocol.ReadGroupSet*

**getReference** (*referenceId*)
Returns the Reference with the specified ID from the server.

**Parameters** **referenceId** (*str*) – The ID of the Reference of interest.

**Returns** The Reference of interest.

**Return type** *ga4gh.protocol.Reference*

**getReferenceSet** (*referenceSetId*)
Returns the ReferenceSet with the specified ID from the server.

**Parameters** **referenceSetId** (*str*) – The ID of the ReferenceSet of interest.

**Returns** The ReferenceSet of interest.

**Return type** *ga4gh.protocol.ReferenceSet*

**getVariant** (*variantId*)
Returns the Variant with the specified ID from the server.

**Parameters** **variantId** (*str*) – The ID of the Variant of interest.

**Returns** The Variant of interest.

**Return type** *ga4gh.protocol.Variant*

**getVariantSet** (*variantSetId*)
Returns the VariantSet with the specified ID from the server.

**Parameters** **variantSetId** (*str*) – The ID of the VariantSet of interest.

**Returns** The VariantSet of interest.

**Return type** *ga4gh.protocol.VariantSet*

**searchDatasets**()
Returns an iterator over the Datasets on the server.

**Returns** An iterator over the *ga4gh.protocol.Dataset* objects on the server.

**searchReadGroupSets**(*datasetId*, *name=None*)
Returns an iterator over the ReadGroupSets fulfilling the specified conditions from the specified Dataset.

**Parameters** **name** (*str*) – Only ReadGroupSets matching the specified name will be returned.

**Returns** An iterator over the *ga4gh.protocol.ReadGroupSet* objects defined by the query parameters.

**Return type** iter

**searchReads**(*readGroupIds*, *referenceId=None*, *start=None*, *end=None*)
Returns an iterator over the Reads fulfilling the specified conditions from the specified ReadGroupIds.

**Parameters**

- **readGroupIds** (*str*) – The IDs of the *ga4gh.protocol.ReadGroup* of interest.
- **referenceId** (*str*) – The name of the *ga4gh.protocol.Reference* we wish to return reads mapped to.
- **start** (*int*) – The start position (0-based) of this query. If a reference is specified, this defaults to 0. Genomic positions are non-negative integers less than reference length. Requests spanning the join of circular genomes are represented as two requests one on each side of the join (position 0).
- **end** (*int*) – The end position (0-based, exclusive) of this query. If a reference is specified, this defaults to the reference's length.

**Returns** An iterator over the *ga4gh.protocol.ReadAlignment* objects defined by the query parameters.

**Return type** iter

**searchReferenceSets**(*accession=None*, *md5checksum=None*, *assemblyId=None*)
Returns an iterator over the ReferenceSets fulfilling the specified conditions.

**Parameters**

- **accession** (*str*) – If not null, return the reference sets for which the *accession* matches this string (case-sensitive, exact match).
- **md5checksum** (*str*) – If not null, return the reference sets for which the *md5checksum* matches this string (case-sensitive, exact match). See `ga4gh.protocol.ReferenceSet::md5checksum` for details.
- **assemblyId** (*str*) – If not null, return the reference sets for which the *assemblyId* matches this string (case-sensitive, exact match).

**Returns** An iterator over the *ga4gh.protocol.ReferenceSet* objects defined by the query parameters.

**searchReferences**(*referenceSetId*, *accession=None*, *md5checksum=None*)
Returns an iterator over the References fulfilling the specified conditions from the specified Dataset.

**Parameters**

- **referenceSetId** (*str*) – The ReferenceSet to search.
- **accession** (*str*) – If not None, return the references for which the *accession* matches this string (case-sensitive, exact match).
- **md5checksum** (*str*) – If not None, return the references for which the *md5checksum* matches this string (case-sensitive, exact match).

**Returns** An iterator over the *ga4gh.protocol.Reference* objects defined by the query parameters.

**searchVariantSets**(*datasetId*)
Returns an iterator over the VariantSets fulfilling the specified conditions from the specified Dataset.

**Parameters** **datasetId** (*str*) – The ID of the *ga4gh.protocol.Dataset* of interest.

**Returns** An iterator over the *ga4gh.protocol.VariantSet* objects defined by the query parameters.

**searchVariants**(*variantSetId*, *start=None*, *end=None*, *referenceName=None*, *callSetIds=None*)
Returns an iterator over the Variants fulfilling the specified conditions from the specified VariantSet.

**Parameters**

- **variantSetId** (*str*) – The ID of the *ga4gh.protocol.VariantSet* of interest.
- **start** (*int*) – Required. The beginning of the window (0-based, inclusive) for which overlapping variants should be returned. Genomic positions are non-negative integers less than reference length. Requests spanning the join of circular genomes are represented as two requests one on each side of the join (position 0).
- **end** (*int*) – Required. The end of the window (0-based, exclusive) for which overlapping variants should be returned.
- **referenceName** (*str*) – The name of the *ga4gh.protocol.Reference* we wish to return variants from.
- **callSetIds** (*list*) – Only return variant calls which belong to call sets with these IDs. If an empty array, returns variants without any call objects. If null, returns all variant calls.

**Returns** An iterator over the *ga4gh.protocol.Variant* objects defined by the query parameters.

**Return type** iter

## 1.5 Configuration

The GA4GH reference server has two basic elements to its configuration: the *Data hierarchy* and the *Configuration file*.

### 1.5.1 Data hierarchy

Data is input to the GA4GH server as a directory hierarchy, in which the structure of data to be served is represented by the file system. At the top level of the data hierarchy there are two required directories to hold the top level container types: `referenceSets` and `datasets`.

**Todo**

We need to link to the high-level API documentation for descriptions of what the various objects here mean.

### ReferenceSets

Within the data directory there must be a directory called `referenceSets`. Within this directory, each directory is interpreted as containing a `ReferenceSet` with the directory name mapped to the name of the reference set. Here is an example of how reference data should be arranged:

```
references/
    GRCh37.json
    GRCh37/
        1.fa.gz
        1.fa.gz.fai
        1.json
        2.fa.gz
        2.fa.gz.fai
        2.json
        # More references
    GRCh38.json
    GRCh38/
        1.fa.gz
        1.fa.gz.fai
        1.json
        2.fa.gz
        2.fa.gz.fai
        2.json
        # More references
```

In this example we have two reference sets, with names `GRCh37` and `GRCh38`. Each reference set directory must be accompanied by a file in JSON format, which lists the metadata for a given reference. For example, the `GRCh37.json` file above might look something like

```
{
    "description": "GRCh37 primary assembly",
    "sourceUri": "TODO",
    "assemblyId": "TODO",
    "sourceAccessions": [],
    "isDerived": false,
    "ncbiTaxonId": 9606
}
```

Within a reference set directory is a set of files defining the references themselves. Each reference object corresponds to three files: the bgzip compressed FASTA sequences, the FAI index and a JSON file providing the metadata. There must be exactly one sequence per FASTA file, and the sequence ID in the FASTA file must be equal to the reference name (i.e., the first line in `1.fa` above should start with `>1`.)

The JSON metadata required for a reference is similar to a reference set. An example might look something like:

```
{
    "sourceUri": "TODO",
    "sourceAccessions": [
        "CM000663.2"
    ],
    "sourceDivergence": null,
    "md5checksum": "bb07c91cda4645ad8e75e375e3d6e5eb",
    "isDerived": false,
    "ncbiTaxonId": 9606
}
```

**Note:** This input format is highly prescriptive and inflexible. Expecting users to calculate md5checksums, in partic-

ular, is unreasonable. A command line utility to import references from a variety of sources is envisaged to take the tedium out of this process and to ensure that the references are correctly set up and indexed.

---

### Datasets

The main container for genetic data is the dataset. Within the main data directory there must be a directory called `datasets`. Within this directory each subdirectory is interpreted as a dataset of that name. For example, we might have something like:

```
datasets/
    1kg-phase1
        variants/
            # Variant data
        reads/
            # Read data
    1kg-phase3
        variants/
            # Variant data
        reads/
            # Read data
```

In this case we specify two datasets with name equal to `1kg-phase1` and `1kg-phase3`. These directories contain the read and variant data within the `variants` and `reads` directory, respectively.

### Variants

Each dataset can contain a number of VariantSets, each of which basically corresponds to a VCF file. Because VCF files are commonly split by chromosome a VariantSet can consist of many VCF files that have consistent metadata. Within the `variants` directory, each directory is interpreted as a variant set with that name. A variant set directory then contains one or more indexed VCF/BCF files.

### Reads

A dataset can contain many ReadGroupSets, and each ReadGroupSet contains a number of ReadGroups. The `reads` directory contains a number of BAM files, each of which corresponds to a single ReadGroupSet. ReadGroups are then mapped to the ReadGroups that we find within the BAM file.

### Example

An example layout might look like:

```
ga4gh-data/
    referencesSet/
        referenceSet1.json
        referenceSet1/
            1.fa.gz
            1.fa.gz.fai
            1.json
            2.fa.gz
            2.fa.gz.fai
            2.json
            # More references
    datasets/
```

```
        dataset1/
            /variants/
                variantSet1/
                    chr1.vcf.gz
                    chr1.vcf.gz.tbi
                    chr2.vcf.gz
                    chr2.vcf.gz.tbi
                    # More VCFs
                variantSet2/
                    chr1.bcf
                    chr1.bcf.csi
                    chr2.bcf
                    chr2.bcf.csi
                    # More BCFs
            /reads/
                sample1.bam
                sample1.bam.bai
                sample2.bam
                sample2.bam.bai
                # More BAMS
```

### 1.5.2 Configuration file

The GA4GH reference server is a Flask application and uses the standard Flask configuration file mechanisms. Many configuration files will be very simple, and will consist of just one directive instructing the server where to look for data; for example, we might have

```
DATA_SOURCE = "/path/to/data/root"
```

For production deployments, we shouldn't need to add any more configuration than this, as the other keys have sensible defaults. However, all of Flask's builtin configuration values are supported, as well as the extra custom configuration values documented here.

When debugging deployment issues, it can be very useful to turn on extra debugging information as follows:

```
DEBUG = True
```

**Warning:** Debugging should only be used temporarily and not left on by default.

#### Configuration Values

**DEFAULT_PAGE_SIZE** The default maximum number of values to fill into a page when responding to search queries. If a client does not specify a page size in a query, this value is used.

**MAX_RESPONSE_LENGTH** The approximate maximum size of a response sent to a client in bytes. This is used to control the amount of memory that the server uses when creating responses. When a client makes a search request with a given page size, the server will process this query and incrementally build a response until (a) the number of values in the page list is equal to the page size; (b) the size of the serialised response in bytes is >= MAX_RESPONSE_LENGTH; or (c) there are no more results left in the query.

**REQUEST_VALIDATION** Set this to True to strictly validate all incoming requests to ensure that they conform to the protocol. This may result in clients with poor standards compliance receiving errors rather than the expected results.

**RESPONSE_VALIDATION** Set this to True to strictly validate all outgoing responses to ensure that they conform to the protocol. This should only be used for development purposes.

**OIDC_PROVIDER** If this value is provided, then OIDC is configured and SSL is used. It is the URI of the OpenID Connect provider, which should return an OIDC provider configuration document.

**OIDC_REDIRECT_URI** The URL of the redirect URI for OIDC. This will be something like `https://SERVER_NAME:PORT/oauth2callback`. During testing (and particularly in automated tests), if TESTING is True, we can have this automatically configured, but this is discouraged in production, and fails if TESTING is not True.

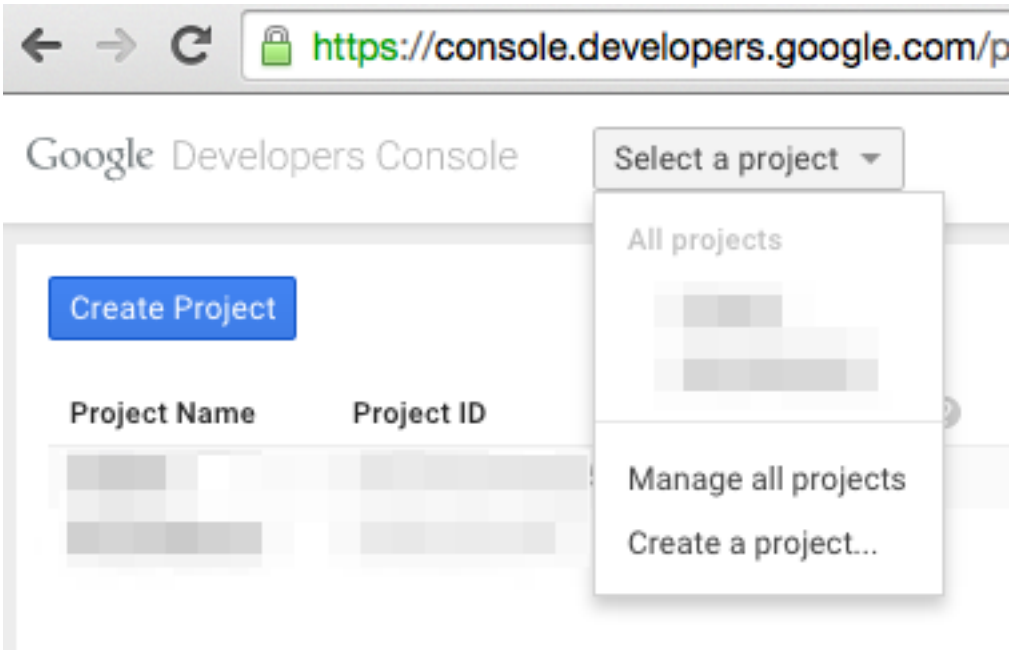
**OIDC_CLIENT_ID, OIDC_CLIENT_SECRET** These are the client id and secret arranged with the OIDC provider, if client registration is manual (google, for instance). If the provider supports automated registration they are not required or used.

**OIDC_AUTHZ_ENDPOINT, OIDC_TOKEN_ENDPOINT, OIDC_TOKEN_REV_ENDPOINT** If the authorization provider has no discovery document available, you can set the authorization and token endpoints here.

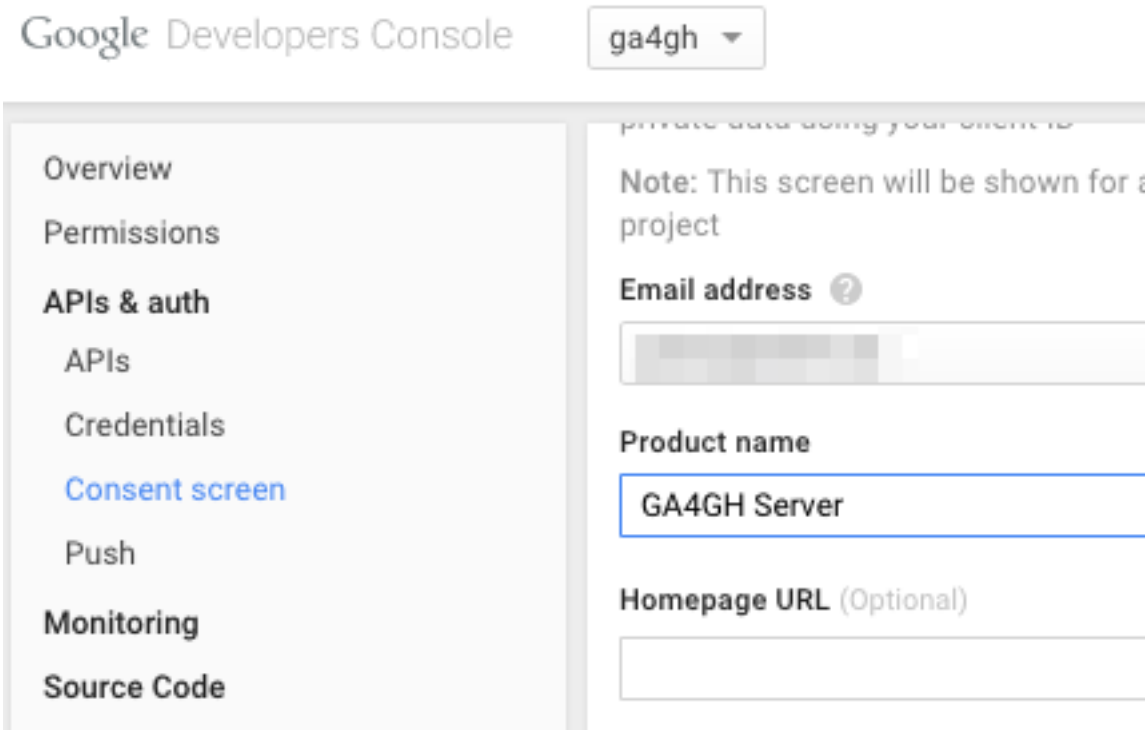
### 1.5.3 OpenID Connect Providers

The server can be configured to use OpenID Connect (OIDC) for authentication. As an example, here is how one configures it to use Google as the provider.

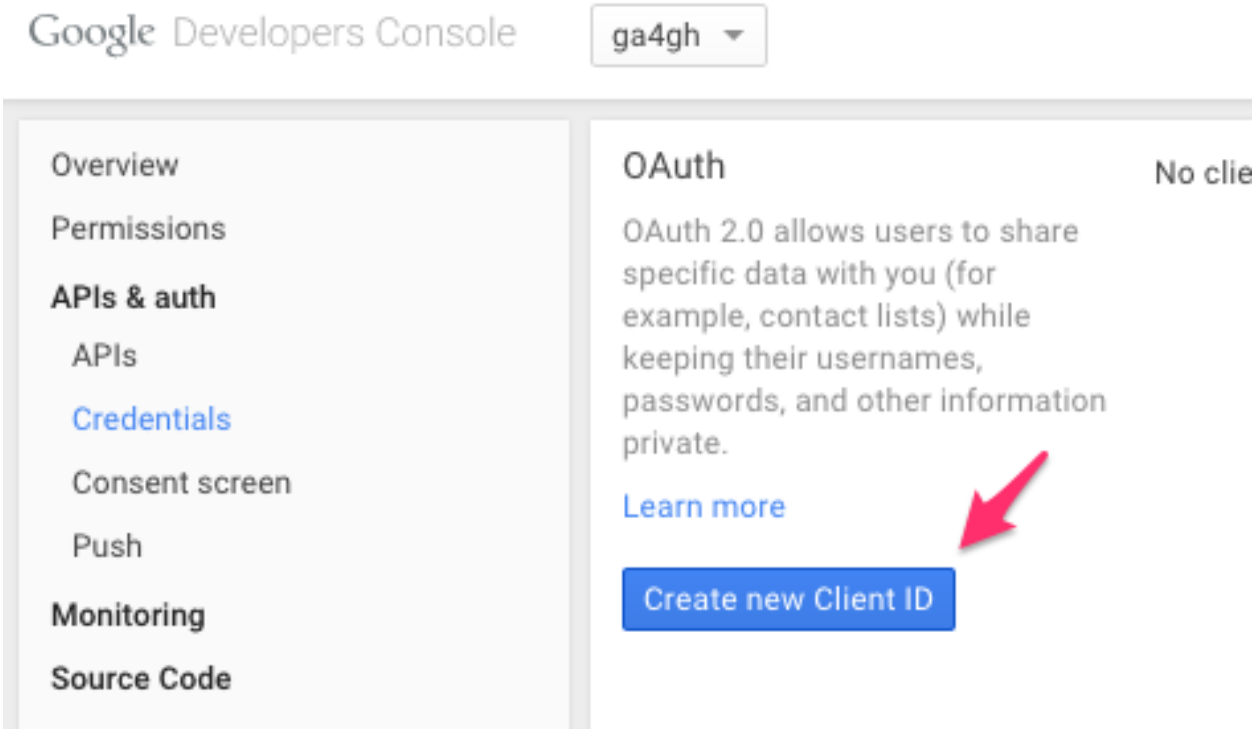
Go to https://console.developers.google.com/project and in create a project.



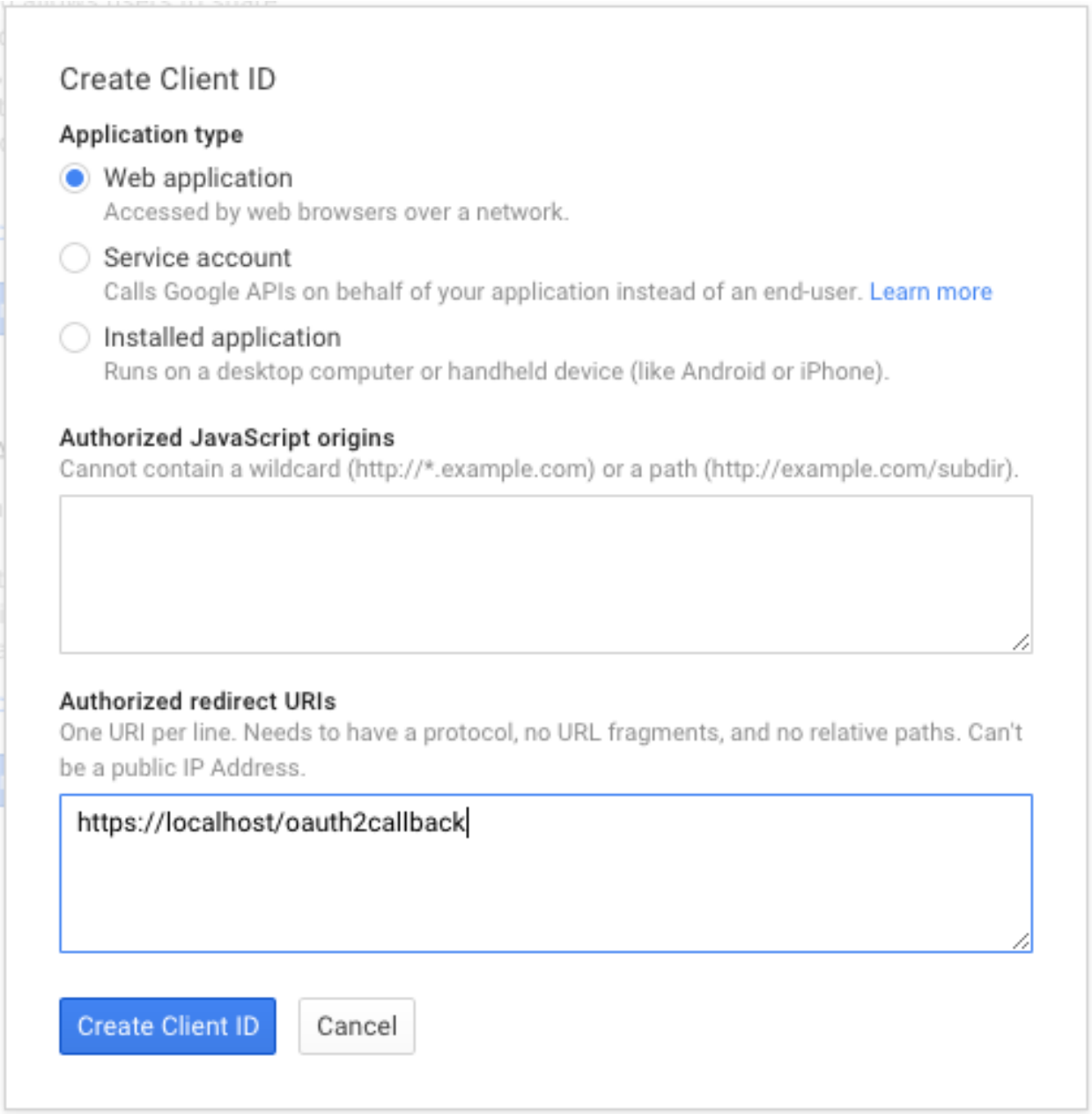

Navigate to the project -> APIs & auth -> Consent Screen and enter a product name



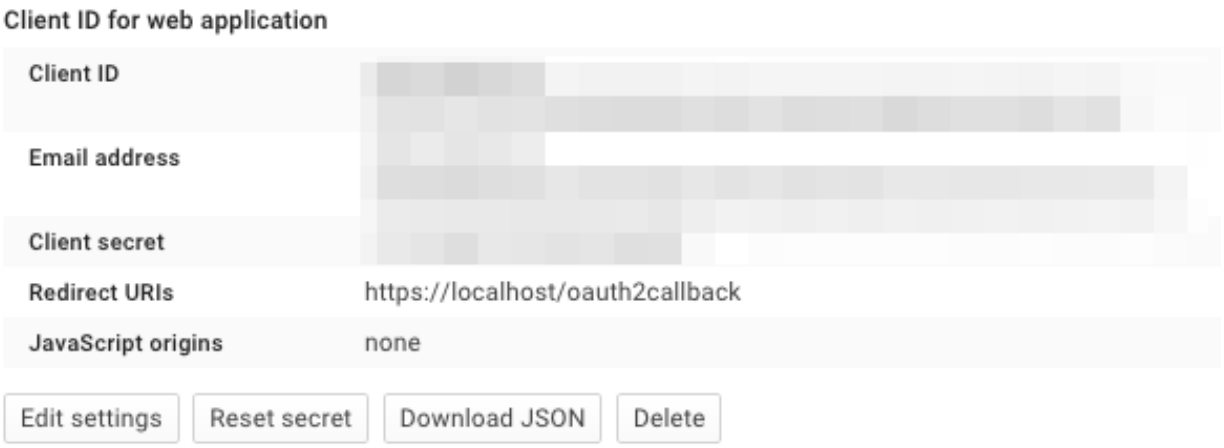

Navigate to project -> APIs & auth -> Credentials, and create a new client ID.




Create the client as follows:




Which will give you the necessary client id and secret. Use these in the OIDC configuration for the GA4GH server, using the *OIDC_CLIENT_ID* and *OIDC_CLIENT_SECRET* configuration variables. The Redirect URI should match the *OIDC_REDIRECT_URI* configuration variable, with the exception that the redirect URI shown at google does not require a port (but the configuration variable does)




## 1.6 Development

Thanks for your interest in helping us develop the GA4GH reference implementation! There are lots of ways to contribute, and it's easy to get up and running. This page should provide the basic information required to get started; if you encounter any difficulties please let us know

**Warning:** This guide is a work in progress, and is incomplete.

### 1.6.1 Development environment

We need a development Python 2.7 installation, Git, and some basic libraries. On Debian or Ubuntu, we can install these using

```
$ sudo apt-get install python-dev git zlib1g-dev libxslt1-dev libffi-dev libssl-dev
```

**Note:** TODO: Document this basic step for other platforms? We definitely want to tell people how to do this with Brew or ports on a Mac.

If you don't have admin access to your machine, please contact your system administrator, and ask them to install the development version of Python 2.7 and the development headers for zlib.

Once these basic prerequisites are in place, we can then bootstrap our local Python installation so that we have all of the packages we require and we can keep them up to date. Because we use the functionality of the recent versions of `pip` and other tools, it is important to use our own version of it and not any older versions that may be already on the system.

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ python get-pip.py --user
```

This creates a user specific site-packages installation for Python, which is based in your `~/.local` directory. This means that you can now install any Python packages you like without needing to either bother your sysadmin or worry about breaking your system Python installation. To use this, you need to add the newly installed version of `pip` to your `PATH`. This can be done by adding something like

```
export PATH=$HOME/.local/bin:$PATH
```

to your `~/.bashrc` file. (This will be slightly different if you use another shell like `csh` or `zsh`.)

We then need to activate this configuration by logging out, and logging back in. Then, test this by running:

```
$ pip --version
pip 6.0.8 from /home/username/.local/lib/python2.7/site-packages (python 2.7)
```

We are now ready to start developing!

### 1.6.2 GitHub workflow

First, go to https://github.com/ga4gh/server and click on the 'Fork' button in the top right-hand corner. This will allow you to create your own private fork of the server project where you can work. See the GitHub documentation for help on forking repositories. Once you have created your own fork on GitHub, you'll need to clone a local copy of this repo. This might look something like:

```
$ git clone git@github.com:username/server.git
```

We can then install all of the packages that we need for developing the GA4GH reference server:

```
$ cd server
$ pip install -r requirements.txt --user
```

This will take a little time as the libraries that we require are fetched from PyPI and built.

It is also important to set up an upstream remote for your repo so that you can sync up with the changes that other people are making:

```
$ git remote add upstream https://github.com/ga4gh/server.git
```

All development is done against the `master` branch.

All development should be done in a topic branch. That is, a branch that the developer creates him or herself. These steps will create a topic branch (replace `TOPIC_BRANCH_NAME` appropriately):

```
$ git fetch --all
$ git checkout master
$ git merge --ff-only upstream/master
$ git checkout -b TOPIC_BRANCH_NAME
```

Topic branch names should include the issue number (if there is a tracked issue this change is addressing) and provide some hint as to what the changes include. For instance, a branch that addresses the (imaginary) tracked issue with issue number #123 to add more widgets to the code might be named `123_more_widgets`.

At this point, you are ready to start adding, editing and deleting files. Stage changes with `git add`. Afterwards, checkpoint your progress by making commits:

```
$ git commit -m 'Awesome changes'
```

(You can also pass the `--amend` flag to `git commit` if you want to incorporate staged changes into the most recent commit.)

Once you have changes that you want to share with others, push your topic branch to GitHub:

```
$ git push origin TOPIC_BRANCH_NAME
```

Then create a pull request using the GitHub interface. This pull request should be against the `master` branch (this should happen automatically).

At this point, other developers will weigh in on your changes and will likely suggest modifications before the change can be merged into `master`. When you get around to incorporating these suggestions, it is likely that more commits

will have been added to the `master` branch. Since you (almost) always want to be developing off of the latest version of the code, you need to perform a rebase to incorporate the most recent changes from `master` into your branch.

> **Warning:** We recommend against using `git pull`. Use `git fetch` and `git rebase` to update your topic branch against mainline branches instead. See the *Git Workflow Appendix* for elaboration.

```
$ git fetch --all
$ git checkout master
$ git merge --ff-only upstream/master
$ git checkout TOPIC_BRANCH_NAME
$ git rebase master
```

At this point, several things could happen. In the best case, the rebase will complete without problems and you can continue developing. In other cases, the rebase will stop midway and report a merge conflict. That is, git has determined that it is impossible for it to determine how to combine the changes from the new commits in the `master` branch and your changes in your topic branch and needs manual intervention to proceed. GitHub has some documentation on how to resolve rebase merge conflicts.

Once you have updated your branch to the point where you think that you want to re-submit the code for other developers to consider, push the branch again, this time using the force flag:

```
$ git push --force origin TOPIC_BRANCH_NAME
```

If you had tried to push the topic branch without using the force flag, it would have failed. This is because non-force pushes only succeed when you are only adding new commits to the tip of the existing remote branch. When you want to do something other than that, such as insert commits in the middle of the branch history (what `git rebase` does), or modify a commit (what `git commit --amend` does) you need to blow away the remote version of your branch and replace it with the local version. This is exactly what a force push does.

> **Warning:** Never use the force flag to push to the `upstream` repository. Never use the force flag to push to the `master`. Only use the force flag on your repository and on your topic branches. Otherwise you run the risk of screwing up the mainline branches, which will require manual intervention by a senior developer and manual changes by every downstream developer. That is a recoverable situation, but also one that we would rather avoid. (Note: a hint that this has happened is that one of the above listed merge commands that uses the `--ff-only` flag to merge a remote mainline branch into a local mainline branch fails.)

One task that you might be asked to do before your topic branch can be merged is "squashing your commits." We want the git history to be clean and informative, and we do that by crafting one and only one commit message per logical change. In the normal course of development (unless one is constantly committing with the `--amend` flag) many intermediate commits can be created that should be squashed down to (usually) one before it can be merged. Do this with (assuming you are in your topic branch):

```
$ git rebase -i `git merge-base master HEAD`
```

This will launch an editor that will give you control over how you want to structure your commits. Usually you just want to "pick" the first commit and "squash" all of the subsequent commits, and then ensure that the final commit message is clean (best practice is to give a short summary of the change on the first line, a blank line, and then a more detailed description of the change following, with the issue number – if there is one – in the detailed description). More information about the interactive rebase process can be found here. Once the commits are to your liking, you can push the branch to your remote repository (which will require a force push if you reordered or deleted commits that existed in the remote version of the branch).

(It usually is a good idea to squash commits before rebasing your topic branch on top of a mainline branch. See the elaboration in the *Git Workflow Appendix* on this topic.)

Once your pull request has been merged into `master`, you can close the pull request and delete the remote branch in the GitHub interface. Locally, run this command to delete the topic branch:

```
$ git branch -D TOPIC_BRANCH_NAME
```

Only the tip of the iceberg of git and GitHub has been covered in this section, and much more can be learned by browsing their documentation. For instance, get help on the `git commit` command by running:

```
$ git help commit
```

To master git, we recommend reading this free book (save chapter four, which is about git server configuration): Pro Git.

### 1.6.3 Contributing

See the files `CONTRIBUTING.md` and `STYLE.md` for an overview of the processes for contributing code and the style guidelines that we use.

### 1.6.4 Development utilities

All of the command line interface utilities have local scripts that simplify development: for example, we can run the local version of the `ga2sam` program by using:

```
$ python ga2sam_dev.py
```

To run the server locally in development mode, we can use the `server_dev.py` script, e.g.:

```
$ python server_dev.py
```

will run a server using the default configuration. This default configuration expects a data hierarchy to exist in the `ga4gh-example-data` directory. This default configuration can be changed by providing a (fully qualified) path to a configuration file (see the *Configuration* section for details).

There is also an OpenID Connect (oidc) provider you can run locally for development and testing. It resides in `/oidc-provider` and has a run.sh file that creates a virtualenv, installs the necessary packages, and runs the server. Configuration files can be found in `/oidc-provider/simple_op`:

```
$ cd oidc-provider
$ ./run.sh
```

The provider expects OIDC redirect URIs to be over HTTPS, so if the ga4gh server is started with OIDC enabled, it defaults to HTTPS. You can run the server against this using:

```
$ python server_dev.py -c LocalOidConfig
```

### 1.6.5 Organisation

The code for the project is held in the `ga4gh` package, which corresponds to the `ga4gh` directory in the project root. Within this package, the functionality is split between the `client`, `server`, `protocol` and `cli` modules. The `cli` module contains the definitions for the `ga4gh_client` and `ga4gh_server` programs.

An important file in the project is `ga4gh/_protocol_definitions.py`. This file defines the classes for the GA4GH protocol. The file is generated using the `scripts/process_schemas.py` script, which takes input data from the GA4GH schemas repo. To generate a new `_protocol_definitions.py` file, use

```
$ python scripts/process_schemas.py -i path/to/schemas desiredVersion
```

Where `desiredVersion` is the version that will be written to the `_protocol_definitions.py` file. This version must be in the form `major.minor.revision` where major, minor and revision can be any alphanumeric string.

### 1.6.6 Git Workflow Appendix

#### Don't use `git pull`

We recommend against using `git pull`. The `git pull` command by default combines the `git fetch` and the `git merge` command. If your local branch has diverged from its remote tracking branch, running `git pull` will create a merge commit locally to join the two branches.

In some workflows, this is not an issue. For us, however, it creates a problem in the future. When you are ready to submit your topic branch in a pull request, we ask you to squash your commits (usually down to one commit). Given the complex graph topography created by all of the merges, the order in which git applies commits in the squash is very difficult to reason about and will likely create merge conflicts that you find unnecessary and nonsensical (and therefore, highly aggravating!).

We instead recommend using `git fetch` and `git rebase` to update your local topic branch against a mainline branch. This will create a linear commit history in your topic branch, which will be easy to squash, since the commits are applied in the squash in the order that you made them.

`git pull` does have the `--rebase` option which will do a rebase instead of a merge to incorporate the remote branch. One can also set the `branch.autosetuprebase always` config option to have `git pull` do a rebase by default (i.e. without passing the `--rebase` flag) instead of a merge. This will avoid the issue of squashing a non-linear commit history.

So, in truth, we are really recommending against squashing local branches with many merge commits in them. However, using the default settings for `git pull` is the easiest way to end up in this situation. Therefore, don't use `git pull` unless you know what you are doing.

#### Squash, then rebase

When updating a local topic branch with changes from a mainline branch, we recommend squashing commits in your topic branch down to one commit before rebasing on top of the mainline branch. The reason for this is that, under the hood, to apply the rebase `git rebase` essentially cherry-picks each commit from your topic branch not in the mainline branch and applies it to the mainline branch. Each one of these applications can cause a merge conflict. It is much better to face the potential of only one merge conflict than N merge conflicts (where N is the number of unique commits in the local branch)!

The difficulty of proceeding the opposite way (rebasing, then squashing) is only compounded because of the unintuitiveness of the N merge conflicts. When presented with a merge conflict, your likely intuition is to put the file in the state that you think it ought to be in, namely the condition it was in after the Nth commit. However, if that state was different than the state that git thinks it should be in – namely, the state of the file at commit X where X<N – then you have only created the potential for more merge conflicts. When the next intermediate commit, Y (where X<Y<N) is applied, it too will create a merge conflict. And so on.

So squash, then rebase, and avoid this whole dilemma. The terms are a bit confusing since both "squashing" and "rebasing" are accomplished via the `git rebase` command. As mentioned above, squash the commits in your topic branch with (assuming you have branched off of the `master` mainline branch):

```
$ git rebase -i `git merge-base master HEAD`
```

(`git merge-base master HEAD` specifies the most recent commit that both `master` and your topic branch share in common. Normally this is equivalent to the most recent commit of `master`, but that's not guaranteed – for

instance, if you have updated your local `master` branch with additional commits from the remote `master` since you created your topic branch which branched off of the local `master`.)

And rebase with (again, assuming `master` as the mainline branch):

```
$ git rebase master
```

#### GitHub's broken merge/CI model

GitHub supports continuous integration (CI) via Travis CI. On every pull request, Travis runs a suite of tests to determine if the PR is safe to merge into the mainline branch that it targets. Unfortunately, the way that GitHub's merge model is structured does not guarantee this property. That is, it is possible for a PR to pass the Travis tests but for the mainline branch to fail them after that PR is merged.

How can this happen? Let's illustrate by example: suppose PR A and PR B both branch off of commit M, which is the most recent commit in the mainline branch. A and B both pass CI, so it appears that it is safe to merge them into the mainline branch. However, it is also true that the changes in A and B have never been tested *together* until CI is run on the mainline branch after both have been merged. If PR A and B have incompatible changes, even if both merge cleanly, CI will fail in the mainline branch.

GitHub could solve this issue by not allowing a PR to be merged unless it both passed CI and its branch contained (in addition to the commits it wanted to merge in to mainline) every commit in the mainline branch. That is, no PR could be merged into mainline unless its commits were tested with every commit already in mainline. Right now GitHub does not mandate this strict sequencing of commits, which is why it can never guarantee that the mainline CI will pass, even if all the PR CIs passed.

Developers could also enforce this property manually, but we have determined that not using GitHub's UI merging features and judiciously re-submitting PRs for additional CI would be more effort than fixing a broken test in a mainline branch once in a while.

### 1.6.7 Release process

There are two types of releases: development releases, and stable bugfix releases. Development releases happen as a matter of course while we are working on a given minor version series, and may be either a result of some new features being ready for use or a minor bugfix. Stable bugfix releases occur when mainline development has moved on to another minor version, and a bugfix is required for the currently released version. These two cases are handled in different ways.

#### Development releases

Version numbers are MAJOR.MINOR.PATCH triples. Minor version increments happen when significant changes are required to the server codebase, which will result in a significant departure from the previously released version, either in code layout or in functionality. During the normal process of development within a minor version series, patch updates are routinely and regularly released.

This entails:

1. Create a PR against `master` with the release notes;
2. Once this has been merged, tag the release on GitHub with the appropriate version number.
3. Fetch the tag from the upstream repo, and checkout this tag. Create the distribution tarball using `python setup.py sdist`, and then upload the resulting tarball to PyPI.
4. Verify that the documentation at http://ga4gh-reference-implementation.readthedocs.org/en/stable/ is for the correct version (it may take a few minutes for this to happen after the release has been tagged on GitHub).

#### Stable bugfix release

When a minor version series has ended because of some significant shift in the server internals, there will be a period when the `master` branch is not in a releasable state. If a bugfix release is required during this period, we create a release using the following process:

1. If it does not already exist, create a release branch called `release-$MAJOR.MINOR` from the tag of the last release.
2. Fix the bug by either cherry picking the relevant commits from `master`, or creating PRs against the `release-$MAJOR.$MINOR` branch if the bug does not apply to `master`.
3. Follow steps 1-4 in the process for *Development releases* above, except using the `release-$MAJOR.$MINOR` branch as the base instead of `master`.

## 1.7 Status

The GA4GH server is currently under active development, and several features are not yet fully functional. These mostly involve the reads API. Some missing features are:

- Unmapped reads. We do not support searching for unmapped reads.
- Searching over multiple ReadGroups.

For more detail on individual development issues, please see the project's issue page.

### 1.7.1 Release Notes

#### 0.2.0

Alpha pre-release supporting major schema update. This release is backwards incompatible with previous releases, and requires a revised data directory layout.

- Schema version changed from v0.5 to v0.6.0a1
- Various backwards incompatible changes to the data directory layout
- Almost complete support for the API.
- Numerous code layout changes.

#### 0.1.2

This bugfix release addresses the following issues:

- #455: bugs in reads/search call (pysam calls not sanitized, wrong number of arguments to getReadAlignments)
- #433: bugs in paging code for reads and variants

#### 0.1.1

- Fixes dense variants not being correctly handled by the server (#391)
- Removes unused paths (thus they won't confusingly show up in the HTML display at / )

### 0.1.0

Just bumping the version number to 0.1.0.

### 0.1.0b1

This is a beta pre-release of the GA4GH reference implementation. It includes

- A fully functional client API;
- A set of client side tools for interacting with any conformant server;
- A partial implementation of the server API, providing supports for variants and reads from native file formats.

### 0.1.0a2

This is an early alpha release to allow us to test the PyPI package and the README. This is not intended for general use.

## A



## C



## D



## E



## F



## G



## H



## I



## L



## M



## N