Global Alliance
for Genomics & Health

GA4GH Documentation

Release 0.1.0a2

Global Alliance for Genomics and Health

May 13, 2015

Contents

This the documentation for the GA4GH reference client and server applications.

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 allows for 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 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
```

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.tar
(ga4gh-env) $ tar -xvf ga4gh-example-data.tar
```

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

```
(ga4gh-env) $ ga4gh_server
* Running on http://0.0.0.0:8000/ (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. 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 '{"datasetIds":[], "name":null}' --header 'Content-Type: application/json' \
http://localhost:8000/v0.5.1/readgroupsets/search
```

In this example, we used the searchReadGroupSets method to ask the server for all the ReadGroupSets on the server. It responded by sending back some JSON, which cURL then printed to the terminal.

Creating these JSON requests by hand is tedious and error prone, and so there is a client application to do this for us. 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 readgroupsets-search http://localhost:8000/v0.5.1
```

The output of this command is a simple summary of the ReadGroupSets that are present on the server. We can also see the JSON messages passing between the client and the server if we increase the verbosity level:

```
(ga4gh-env) $ ga4gh_client -vv readgroupsets-search http://localhost:8000/v0.5.1
```

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/v0.5.1
1kg-phase1
1kg-phase3
```

This tells us that we have two VariantSets on the server, with IDs `1kg-phase1` and `1kg-phase3`. In our example data, these correspond to a subset of the data from 1000 Genomes phases 1 and 3.

We can then search for variants overlapping a given interval in a VariantSet as follows:

```
(ga4gh-env) $ ga4gh_client variants-search http://localhost:8000/v0.5.1 \
--variantSetIds=1kg-phase1 --referenceName=2 --start=33100 --end=34000
```

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-side 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
import ga4gh.protocol as protocol

httpClient = client.HttpClient("http://localhost:8000/v0.5.1")
request = protocol.GASearchVariantsRequest()
request.variantSetIds = ["1kg-phase1"]
request.referenceName = "2"
request.start = 33100
request.end = 34000
for variant in httpClient.searchVariants(request):
    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

1. 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.
2. Update the client API to be more user-friendly. We shouldn't need to create an instance of `GASearchVariantsRequest` to call `searchVariants`. Rather, `searchVariants` should have the corresponding values as parameters which have sensible defaults.

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 zlib1g-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.tar
$ tar -xf ga4gh-example-data.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 Configuration

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

1.4.1 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. For now, we support only one dataset, but this will be generalised to multiple datasets in later releases. An example data layout might be:

```
ga4gh-data/
    /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/
        readGroupSet1
            sample1.bam
            sample1.bam.bai
            sample2.bam
            sample2.bam.bai
            # More BAMS
```

1.4.2 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 all 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.

1.5 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.

1.5.1 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 zlib1g-dev git
```

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!

1.5.2 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 `develop` branch. The `stable` branch is meant to be kept stable since it is the branch releases are based on – don't touch it! These are the two mainline branches. Our branching model is loosely based on the one described here.

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 develop
$ git merge --ff-only upstream/develop
$ 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 `develop` 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 `develop`. When you get around to incorporating these suggestions, it is likely that more commits will have been added to the `develop` 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 `develop` into your branch.

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

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 `develop` 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 `upstream`. Never use the force flag to push to `develop` or `stable`. 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 develop
```

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. Any commit in your topic branch could cause a merge conflict, and it's usually easier to ensure only one merge conflict will potentially occur, rather than performing a merge conflict resolution for each commit in your topic branch – the worst case.)

Once your pull request has been merged into `develop`, 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
```

1.5.3 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.

1.5.4 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).

1.5.5 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.