Lightning Documentation

Release 0.1.0

Curoverse

October 20, 2015

CONTENTS

1 Getting started **3**
1.1 Lightning at a Glance 3
1.2 Importing a Genome 5
1.3 Lightning Design Doc 5
1.4 Installation 7

2 Tiling Overview **9**

3 Compact Genome File (CGF) Format **11**

4 Lantern Specifications **13**

5 Tile Library **15**

6 Annotile: Annotating Tile Variants **17**
6.1 Adding an Annotation Pipeline to Annotile 17
6.2 How does Annotile Work? 17
6.3 Storing Annotation Details 18
6.4 Future Annotile Functionality 18

7 Data Structures Specifications **19**
7.1 Data Structures Specifications, v0.1.0 19
7.2 Data Structures Specifications, v0.1.1 29

8 Lightning API Specifications **39**
8.1 Lightning v0.1.0 API Specifications 39
8.2 Reasoning Behind the API 55
8.3 Lightning Errors 56
8.4 Lightning v0.1.1 API Specifications 56
8.5 Batch Processing 60
8.6 Versioning 61
8.7 Paging 61

9 Software Development Kits **63**

10 Sprite **65**
10.1 Sprite Annotation Database 65

11 Indices and tables **73**

Contents:

CHAPTER

ONE

GETTING STARTED

Welcome to Lightning! This documentation section is most relevant for first time users. Lightning at a Glance provides an overview of Lighting: its uses and components. How-to guides and tutorials are expected to be placed here as the project grows.

1.1 Lightning at a Glance

Lightning is software designed to enable fast queries and machine learning on genomic data. The genomic data we currently focus on are human whole genomes that are aligned and are called by external software, then imported into Lightning. From there, Lighting:

- Stores quality information
- Stores phased and unphased genomes
- Allows fast retrieval of called sequences from regions of interest
- Defines flexible queries:
  - Filters by subsets of the population
  - And/or by specific regions of interest
- Normalizes standard called genome files (such as VCF and gVCF), such that each variant is expressed in the same way
- Incorporates new data fast and painlessly
- Stores annotations from ClinVar and annotation pipelines, such as CAVA

Lightning is made possible by the process of `tiling`, which takes advantage of the high degree of redundancy in a population of genomes. Tiling partitions genomes into `tiles`: overlapping, variable-length sequences that begin and end with unique k-mers, termed `tags`. Once a genome has been tiled, the sequences for each tile are stored in a tile library. These sequences may be annotated by using `Annotile`. The tiled genomes are stored in `Compact Genome Format (CGF)` files. Genomes stored as CGF files are loaded into `Lantern`, which is our in-memory database designed to respond to queries quickly. Finally, `Sprite` is a web browser application for interacting with Lightning.

Note: Stated another way, Lightning's basic method is to consider short snippets of genomic sequences as the basic building block of genomes. These short snippets are of variable length, but are mostly in the range of 250 base pairs long. Splitting genomes into short segments allows for savings by only storing a single copy of redundant sequences.

Each genome is partitioned into these these short read segments. From all *tiles* in a population, a *tile library* can be constructed. Tiles are chosen to have 24mer tags on either end that overlap with neighboring tiles. *Tags* are chosen with with some uniqueness constraint on them and provide convenient anchor points to differentiate tiles from one another.

Currently, all tags are chosen to be at least 2 edit distance away from each other. The tag set is fixed and acts as anchor points to partition future sample genomic sequences wishing to be analyzed.

The hope is that tiles, along with information on the population used to generate them, can also be used to aid in read placement.

Because most genomic sequences are redundant, duplicate tiles need not be stored in a population of genomic sequences. At each tile position, multiple tile variants are stored representing the variation in a population for that tile. Given a partitioned genomic sequence and a tile library, a compact representation of a genome can be constructed by storing the variant numbers contiguously.

1.1.1 Components

- Tiling
- Compact Genome Format (CGF)
- Lantern
- Tile Library
- Annotile
- Sprite
- Software development kits
- API

1.1.2 Motivation

We developed Lightning in response to the difficulty and time-consuming nature of merging VCFs, querying subsets of a population, finding poorly sequenced regions, and similar issues. After using various ad-hoc solutions, we eventually stepped back and committed time and effort to developing a more sensible and sustainable solution. We hope it will be useful to the broader research community and welcome your feedback.

1.1.3 Usage

We host a Lightning instance for public whole genomes, including 502 genomes from the 1000 Genomes Project and 178 genomes from the Harvard Personal Genome Project. Here, the public can interact with Lightning using Sprite.

Todo

Decide where the public lightning instance will be

Currently, Lightning assumes it has access to an Arvados cluster. For a user to run their own instance of Lightning on their data, they must set up an Arvados cluster and import their data into it. From there, they should follow the instructions in Importing Genomes to add new genomes. To add new annotations, they should go to Annotile.

1.1.4 Further research

Lightning is currently in its infancy. This state means Lightning has many use cases and future directions that can be explored, some of which are listed here.

- Implement Lightning Servers for multiple species

- Include RNA sequences in Lightning

Todo

Keep filling out list

1.1.5 Contibutors

- Abram Connelly
- Sarah (Sally) Guthrie
- Jiayong Li
- Nancy Ouyang
- Alexander (Sasha) Wait Zaranek

1.1.6 Contact

lightning@curoverse.com

Todo

Make sure this email is live

1.2 Importing a Genome

Importing a genome is difficult. We'll probably do it using pipelines. Have one pipeline in pipeline_templates that is run when someone uploads a genome.

We acknowledge that phenotypes or meta-information about each genome is vital for interpreting results. However, the breadth of available phenotype databases and the number of possible pieces of information makes using Lightning as a phenotype database infeasible. Sprite provides a preliminary phenotype database, which includes information about whether each genome (or set of calls, also known as a callset) is a reference. Additionally, it stores whether the phases are well known, as well as the sex and the ethnicity.

Todo

Implement a better process for importing a genome Document the process

1.3 Lightning Design Doc

Goal: Specify Lightning v0.1.0 in enough detail that it can be implemented and tested.

1.3.1 Lightning v0.1.0 Features

Features here are expected to only work for well-sequenced tile variants. Poorly sequenced tile variants/partial tile variants should return a null value.

- Stores 1 byte of quality information for all bases (well sequenced versus poorly sequenced)

- Stores human genomes assuming phasing from called genome input
- Retrieves well-sequenced subsequences from population (subsequences identified by tile position)
    - Subsequences may be in the form of bases
    - Subsequences may be in the form of tiles
- Allows filtering
    - by population subsets (named by id)
    - by regions of interest (named by tile)
    - by tile variant (named by md5sum, only well-sequenced tiles)
    - by combinations of the above
- Provides translation between tile positions and reference locations (GRCh37, GRCh38)
- Tiles genomes provided in GFF and gVCF formats
- Can convert tiled genomes to GFF and/or gVCF
- One pipeline is required to add a new genome
- Stores annotations from ClinVar and annotation pipelines, such as CAVA
- Retrieves annotations for one or multiple tile variants (identified by tile variant identifier)
- Retrieves tile variants with one or multiple annotations (identified by annotation identifier)

1.3.2 Lightning v0.1.0 Benchmarking Goals

- Uses up to 70 MB per human genome
- Retrieves the sequences of 50 contiguous tiles from 100 human genomes in 5 seconds
- Retrieves the 50 contiguous tiles from 100 human genomes in less than a second
- Filtering on above query with any option does not change the observable time
- Tiling genomes from GFF/gVCF takes 1 hour per genome
- Annotates the tile library for 100 human genomes in 2 hours
- Filtering on annotations or tile variants for tile library of 100 genomes takes less than 1 second

1.3.3 Lightning v0.1.0 Usage

Lightning v0.1.0 should be run by Curoverse and should include 178 whole genomes from the Personal Genome Project and 502 whole genomes from the 1000 Genomes Project. The tile library should be annotated with CAVA, ClinVar, and ExAC data. The public should be able to retrieve well-sequenced subsequences from this population, filter on them by population subsets, by regions of interest, by tile variant, and/or by annotations.

Users should be able to upload whole genomes, with the understanding that this Lightning instance is public. These newly added genomes should be added to the tile library, new sequences should be annotated with CAVA, ClinVar, and ExAC annotations, and these genomes should be added to those we can filter on.

1.3.4 Lightning v0.1.0 API

See Lightning v0.1.0 API Specifications for API details.

1.3.5 Future Versions Lightning Features

- Support partial tile variants for features detailed in v0.1.0
- Stores unphased and phased human genomes, including phase groups for features detailed in v0.1.0
- Provides translation between tiles and alternate reference locations
- Compare multiple genomes with each other

1.3.6 Future Versions Lightning Usage

Users should set up an Arvados cluster. From there, they should be able to run Sprite, which will allow them to import their data without making it public. From there, users should be able to use Sprite identically to the v0.1.0 Lightning instance.

1.4 Installation

Since Lightning is in development, our installation process has not yet been determined, implemented, or documented.

Todo

Create a good installation process and document it

CHAPTER

TWO

TILING OVERVIEW

Lightning is made possible by the process of `tiling`, which takes advantage of the high degree of redundancy in a population of genomes. Tiling partitions genomes into `tiles`: overlapping, variable-length sequences that begin and end with unique k-mers, termed `tags`. This document is an overview of our tiling implementation, along with the crunch scripts and pipeline templates available for tiling genome inputs.

We currently accept GFF files and Complete Genomics CGI-var files as inputs.

Todo

- Document PASTA/FASTA
- Write basic instructions for writing one's own conversion of file to a tiling (As I understand it, one must write a conversion between that file format and PASTA, then probably add a pipeline template component).

CHAPTER

THREE

COMPACT GENOME FILE (CGF) FORMAT

Compact Genome File Format is our preliminary format that stores tiled genomes compactly.

Todo

Document details about CGF format

CHAPTER

FOUR

LANTERN SPECIFICATIONS

Lantern is our in-memory database which stores individual tiled genome sequences. It allows us to make very fast comparisons between genomes and filter on genome sequences. Since it is contained in random access memory (RAM), queries are very fast.

Todo

Document Lantern and the REST APIs used to interact with Lantern

CHAPTER

FIVE

TILE LIBRARY

The tile library is our in-memory database that stores tile variants and their information. Tile variants are associated with their sequence, MD5 hash digest, tile position, and other meta-information. The tile library stores this information compactly while enabling queries on it.

Todo

Describe/specify the queries the Tile Library supports

CHAPTER

SIX

ANNOTILE: ANNOTATING TILE VARIANTS

6.1 Adding an Annotation Pipeline to Annotile

Build a component for the *annotate_tile_variants* pipeline. This is likely to require implementing a crunch script and creating a docker image.

6.1.1 Automated Annotation Pipelines

If the annotation pipeline you wish to port cannot use a VCF file (aligned against GRCh37) as input, you will also need to implement a component to generate the required input for your annotation pipeline. Use the *create-vcf-per-tile* component as an example.

The components you implement must output a collection with a directory named *annotile_input*.

Todo

After implementing annotile, document the output format requirements

6.1.2 User-added Annotations

User added annotations can be converted into a component in the *annotate_tile_variants* pipeline by hard-coding annotations applying to specific tile variants. A component which does this already is linked to a Sprite user-added annotations app. Use this component and application as an example.

Annotating genomes is an essential piece of bioinformatic workflows. Annotations are produced by many sources, including programs and users, and we want to ensure Lightning supports the addition of all these annotations to tile variants, can add software-produced annotations to new tile variants automatically, and can respond to queries about which tile variants have which annotations or which annotations apply to particular tile variants.

Despite the importance of annotations in bioinformatics, Lightning is not meant to be an annotation database - it is not meant to support complex queries on types of annotations, source of annotations, date of generation or modification, or keywords. These types of queries and annotation storage should be done by add-ons, which can be added to Sprite using django apps or can be added to one's own application interacting with Lightning.

6.2 How does Annotile Work?

Lightning associates each tile variant with its `variant value`, which is our term for the MD5 hash digest of its sequence. These `variant values` can be associated with metadata about that tile variant. Annotile is simply a many-to-many database, which associates variant values to user-specified annotations identifiers.

Warning: Lightning v0.1.0 does not support annotating tiles with poorly sequenced regions. These require more complex tile variant representations and are thus left to future versions.

Annotile is populated by running the pipeline template *annotate_tile_variants*. Each component in *annotate_tile_variants* runs annotation software on each tile variant, producing detailed annotation information for the annotation app and information for loading into Annotile.

For details about porting an annotation pipeline or adding user-written annotations, see Adding an Annotation Pipeline to Annotile.

Annotile also provides querying capabilities, allowing a user to find which annotations are associated with which tile variants and vice versa. For specific details about annotation querying, see *../api/index*.

6.3 Storing Annotation Details

Sprite has a basic app for storing and visualizing specific annotations, but it is not the only option for storing and querying annotations, since we are designing Lightning to be able to support new annotation databases. A user can build a django app to plug into Sprite on their Lightning instance, or a user can build their own application in their language of choice to work similarly to Sprite, then plug their annotation database into that.

6.4 Future Annotile Functionality

Future functionality is expected to focus on Annotile's querying capabilities, and will probably include a direct querying function from genomes to annotations and vice versa.

CHAPTER

SEVEN

DATA STRUCTURES SPECIFICATIONS

Here are some data structures.

Contents:

7.1 Data Structures Specifications, v0.1.0

7.1.1 ArvadosUUID

The UUID of an Arvados object.

```
'<string>-<string>-<string>'
```

Example ArvadosUUID:

```
'su92l-d1hrv-1xal9oo1iyi7tzn'
```

7.1.2 TilePosition

A compact representation of a tile position. A string of 3 period-separated integers (in base 16). The first integer is the tag set version integer, the second is the path number, the third is the step.

```
'<int>.<int>.<int>'

Version.Path.Step
```

Example TilePosition:

```
'00.2c5.00a1'
```

7.1.3 TilePositionRange

A compact representation of a range of tile positions. A string of 3 period-separated integers (in base 16) followed by a hyphen and another integer in base 16. The first integer is the tag set version integer, the second is the path integer, the third is the first step integer, which is the step to start retrieving from (inclusive and 0-indexed). The integer following the hyphen indicates the step to stop retrieving from (**exclusive** and 0-indexed).

```
'<int>.<int>.<int>-<int>'

Version.Path.StartStep-EndStep
    StartStep: inclusive, 0-indexed
    EndStep: exclusive, 0-indexed
```

Example TilePositionRange:

```
'00.2c5.00a1-00b0'
```

7.1.4 TileVariant

A compact representation of a tile variant. A string of 3 period-separated integers (in base 16), followed by another period and one string. The first integer is the tag set version integer, the second is the path integer, the third is the step integer, and the fourth is the MD5 hash digest of the tile variant sequence.

```
'<int>.<int>.<int>.<string>'

Version.Path.Step.VariantMD5SUM
```

Example TileVariant:

```
'00.247.1bfb.c95325c08a449529143776e18561db71'
```

7.1.5 NotTileVariant

Only used to build *TileVariantClause*. When used, it indicates a selection on the specimens that do not have that tile variant. A *TileVariant* (in base 16), preceded by a tilda.

```
'~<int>.<int>.<int>.<string>'

Version.Path.Step.VariantMD5SUM
```

Example NotTileVariant:

```
'~00.247.1bfb.c95325c08a449529143776e18561db71'
```

7.1.6 ClauseEntry

Used as a builder for *TileVariantClause*'s. Items may be of type *TileVariant* or *NotTileVariant*. If the item is of type *TileVariant*, the *ClauseEntry* evaluates to true if that *TileVariant* exists in the population. If the item is of type *NotTileVariant*, the *ClauseEntry* evaluates to true if the *TileVariant* following the tilda does *not* exist in the population.

```
<TileVariant> | <NotTileVariant>
```

Example ClauseEntry:

```
'00.247.1bfb.c95325c08a449529143776e18561db71'
```

7.1.7 TileVariantClause

The 'OR' list of SAT (Boolean Satisfiability Problem). Ensuring the validity of the clause is a task placed on the client. A list of length 1 or more. Each item in the list is of type *ClauseEntry* and is checked against each phase independently of the other items. The *TileVariantClause* evaluates to true if *any* of the clauses evaluate to true.

```
[ <ClauseEntry>, <ClauseEntry>, ... ]
```

Example TileVariantClause:

```
[
  '00.247.1bfb.c95325c08a449529143776e18561db71',
  '~00.2c5.0000.1948117b4a56e4ad73d36dce185110fd'
]
```

This example will evaluate to True for any genomes in the population that have tile variant `c95325c08a449529143776e18561db71` at *TilePosition* 00.247.1bfb on at least one of their phases and/or do not have tile variant `1948117b4a56e4ad73d36dce185110fd` at *TilePosition* 00.2c5.0000 on at least one of their phases.

7.1.8 TileVariantLogic

The 'AND' list of SAT (Boolean Satisfiability problem). Ensuring the validity of the clause is a task placed on the client. A list of one or more *TileVariantClause*'s. By default, each *TileVariantClause* is evaluated against each phase independently of the other *TileVariantClause*'s.

```
[<TileVariantClause>, <TileVariantClause>, ... ]
```

Example TileVariantLogic:

```
[[
  '00.247.1bfb.c95325c08a449529143776e18561db71',
  '~00.2c5.0000.1948117b4a56e4ad73d36dce185110fd'
]]
```

This example will evaluate to True for any genomes in the population that have tile variant `c95325c08a449529143776e18561db71` at *TilePosition* 00.247.1bfb on at least one of their phases and/or do not have tile variant `1948117b4a56e4ad73d36dce185110fd` at *TilePosition* 00.2c5.0000 on at least one of their phases.

Example TileVariantLogic:

```
[
  ['00.247.1bfb.c95325c08a449529143776e18561db71'],
  ['~00.2c5.0000.1948117b4a56e4ad73d36dce185110fd']
]
```

This example will evaluate to True for any genomes in the population that have tile variant `c95325c08a449529143776e18561db71` at *TilePosition* 00.247.1bfb on at least one of their phases *and* do not have variant `1948117b4a56e4ad73d36dce185110fd` at *TilePosition* 00.2c5.0000 on at least one of their phases.

7.1.9 TileVariantDetail

The metadata information associated with one well sequenced tile variant. Dictionary containing the keys:

- `tile-variant`: The *TileVariant* identifier.
- `tag-length`: The length of the tags.

- `start-tag`: The sequence of the start tag. Must be of length 0 (if the tile is at the start of the path) or the length specified by `tag-length`. Cannot have n's.
- `end-tag`: The sequence of the end tag. Must be of length 0 (if the tile is at the end of the path) or the length specified by `tag-length`. Cannot have n's.
- `is-start-of-path`: A boolean indicating whether the tile is at the start of the path.
- `is-end-of-path`: A boolean indicating whether the tile is at the end of the path.
- `sequence`: The sequence of the tile. Cannot include n's (since the tile variant must be well-sequenced).
- `md5sum`: The md5sum of the tile sequence.
- `length`: The length of the tile sequence.
- `number-of-positions-spanned`: The number of tile positions this tile variant spans. Must be greater or equal to 1.
- `population-frequency`: The percentage of the population that contains this tile variant. Each well-sequenced phase counts as 1 entry in the population.
- `population-count`: The number of well-sequenced phases that contain this tile variant.
- `population-total`: The number of phases that contain a well sequenced tile at this tile position. Each phase counts as 1 entry in the population.

```
{
    'tag-length': <int>,
    'start-tag': <string>,
    'end-tag': <string>,
    'is-start-of-path': <boolean>,
    'is-end-of-path': <boolean>,
    'sequence' : <string>,
    'md5sum': <string>,
    'length': <int>,
    'number-of-positions-spanned': <int>,
    'population-frequency': <float>,
    'population-count': <int>,
    'population-total': <int>
}
```

Validations for each key are as follows:

```
TileVariantDetail['tag-length'] >= 1
TAG_LENGTH = TileVariantDetail['tag-length']
TileVariantDetail['start-tag'] matches '^[acgt]{TAG_LENGTH}$|^$'
TileVariantDetail['end-tag'] matches '^[acgt]{TAG_LENGTH}$|^$'
TileVariantDetail['sequence'] matches '[acgt]'
TileVariantDetail['md5sum'] == MD5_hash_digest(TileVariantDetail['sequence'])
TileVariantDetail['length'] == len(TileVariantDetail['sequence'])
TileVariantDetail['number-of-positions-spanned'] >= 1
0 <= TileVariantDetail['population-frequency'] <= 1
TileVariantDetail['population-total'] >= 0
```

Example TileVariantDetail:

```
{
    'tile-variant':'00.2c5.30ae.bc952f709d7419f7e103daa2b7e469a9',
    'tag-length': 24,
    'start-tag': 'gccaaggagttttaaaactactga',
    'end-tag': '',
    'is-start-of-path': False,
    'is-end-of-path': True,
    'sequence' : 'gccaaggagttttaaaactactgatgcccacctcccacacccaaaagtctgattaattgatctagggtatggcctgag
    'md5sum': 'bc952f709d7419f7e103daa2b7e469a9',
    'length': 394,
    'number-of-positions-spanned': 1,
    'population-frequency': 0.5,
    'population-count': 150,
    'population-total': 300
}
```

7.1.10 Assembly

A description of an assembly (JSON-formatted). Keys:

- `assembly-name`: the assembly name (string)
- `assembly-pdh`: the portable data hash referencing the collection of FASTA files for this assembly (string)

```
{
  'assembly-name': <string>,
  'assembly-pdh': <string>,
}
```

Example Assembly:

```
{
    'assembly-name': 'hg19',
    'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735'
}
```

7.1.11 Locus

A description of an assembly locus (JSON-formatted). Keys:

- `assembly-name`: the assembly name (string)
- `assembly-pdh`: the portable data hash referencing the collection of FASTA files for this assembly (string)
- `chromosome-name`: the chromosome name (string)
- `indexing`: Indicates the indexing of `start-position` and `end-position`. (hard-coded to 0).
- `start-position`: start position; the inclusive beginning of the loci range. Must be greater than 0 and less than `end-position`. Inclusive. (Integer)
- `end-position`: end position; the exclusive end of the loci range. Must be greater than `start-position` and less than or equal to the length of the chromosome in the specified assembly.

```
{
  'assembly-name': <string>,
  'assembly-pdh': <string>,
  'chromosome-name': <string>,
  'indexing': 0,
  'start-position': <int>,
  'end-position': <int>
}
```

Example Locus:

```
{
    'assembly-name': 'hg19',
    'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735',
    'chromosome-name': '13',
    'indexing': 0,
    'start-position': 32199976,
    'end-position': 32200225
}
```

7.1.12 CMPFunction

List of supported comparison functions.

```
ENUM('eq', 'lt', 'lte', 'gt', 'gte')
```

Example CMPFunction:

```
'eq'
```

7.1.13 CMPTuple

Representation of an integer comparison - used for filtering queries. Also includes a range comparison. For the range comparison, the first integer is the inclusive beginning of the range, the second int is the exclusive end of the range.

```
( <CMP-fn>, <float>)
OR
('range', <float>, <float>)
```

Example CMPFunction:

```
('lt', 5)
```

7.1.14 VCF Data Structures Specifications

gVCFBlock

Genotype quality ranges used for banding. List of integers (length greater than or equal to 2). Each pair of integers creates a minGQ (inclusive) and maxGQ (exclusive) pair for a gVCFBlock. Must be strictly increasing, the first entry must be 0, and the last entry must be 2147483647 (the largest unsigned integer representable in 32 bits).

```
[0, <int>, ..., 2147483647]
```

Example gVCFBlock:

```
[0, 2147483647]
```

gVCFMetaData

Representation of a gVCF Header. JSON-formatted with keys:

- `fileformat`, whose value indicates the format of the VCFLine's returned by the Lightning server
- `fileDate`, the current date, format (YYYYMMDD)
- `source`, the Lightning server producing the VCF MetaData
- `assembly`, *Assembly* data type containing the location of the reference FASTA file used to generate the VCF lines.
- `info`, used to indicate the end of a VCFLine.
- `format`, for genotype fields
- `alt`, used to refer to non-reference alternate alleles.
- [optional] `gvcfblock`, for information about splitting gVCF blocks

```
{
    'fileformat':'VCFc4.2',
    'fileDate':<int>,
    'source':'Lightningv0.1.0',
    'assembly':<Assembly>,
    'info': [
      {
        'ID': 'END',
        'Number':1,
        'Type':Integer,
        'Description':'Stop position of the interval'
      }
    ],
    'format': [
      {
        'ID': 'GT',
        'Number':1,
        'Type':'String',
        'Description':'Genotype'
      }
    ],
    'alt': [
      {
        'ID': 'NOT_REF',
        'Description':'Represents any possible alternative allele at this location'
      }
    ],
    'gvcfblock': gVCFBlock_
}
```

Example gVCFMetaData:

```
{
    'fileformat':'VCFc4.2',
    'fileDate':20150928,
    'source':'Lightningv0.1.0',
    'assembly':'dad94936d4144f5e0a289244d8be93e9+5735/hg19',
    'info': [
      {
        'ID': 'END',
        'Number':1,
        'Type':Integer,
        'Description':'Stop position of the interval'
      }
    ],
    'format': [
      {
        'ID': 'GT',
        'Number':1,
        'Type':'String',
        'Description':'Genotype'
      }
    ],
    'alt': [
      {
        'ID': 'NOT_REF',
        'Description':'Represents any possible alternative allele at this location'
      }
    ],
    'gvcfblock': [0, 2147483647]
}
```

VCFMetaData

Representation of a VCF Header. JSON-formatted with keys:

- `fileformat`, whose value indicates the format of the VCFLine's returned by the Lightning server
- `fileDate`, the current date, format (YYYYMMDD)
- `source`, the Lightning server producing the VCF MetaData
- `assembly`, *Assembly* data type containing the location of the reference FASTA file used to generate the VCF lines.
- `format`, for genotype fields

```
{
  'fileformat':'VCFc4.2',
  'fileDate':<int>,
  'source':'Lightningv0.1.0',
  'assembly':<Assembly>,
  'format': [
    {
      'ID': 'GT',
      'Number':1,
      'Type':'String',
      'Description':'Genotype'
    }
  ]
}
```

Example VCFMetaData:

```
{
    'fileformat':'VCFc4.2',
    'fileDate':20150928,
    'source':'Lightningv0.1.0',
    'assembly':'dad94936d4144f5e0a289244d8be93e9+5735/hg19',
    'format': [
      {
        'ID': 'GT',
        'Number':1,
        'Type':'String',
        'Description':'Genotype'
      }
    ]
}
```

VCFSampleFormatData

JSON-formatted format field format for a VCF or gVCF file. Used for Genotype fields. Keys:

- `sample-name`, the name of the sample, normally defined by the column header
- `GT`, the genotype of the sample

```
{
  'sample-name': <string>,
  'GT':<string>
}
```

Example VCFSampleFormatData:

```
{'sample-name':'human1-illumina', 'GT':'0/0'}
```

VCFLine

Representation of VCF line. Keys:

- `chrom`: Chromosome. An identifier from the reference genome or an angle-bracketed ID String (<ID>) pointing to a contig in the assembly file (the file pointed to by `assembly` in *VCFMetaData*). The colon symbol (:) must be absent from all chromosome names to avoid parsing errors when dealing with breakends. (String, no white-space permitted).
- `pos`: Position. The reference position, 1-indexed. Telomeres are indicated by using positions 0 or N+1, where N is the length of the corresponding chromosome or contig. (Integer).
- `ref`: Reference base(s). Each base must be one of A,C,G,T,N (case insensitive). Multiple bases are permitted. The value matching the `pos` key refers to the position of the first base in this string. For simple insertions and deletions in which either the `ref` or one of the `alt` alleles would otherwise be null/empty, the `ref` and `alt` values must include the base before the event (which must be reflected in the `pos` field), unless the event occurs at position 1 on the contig in which case it must include the base after the event; this padding base is not required (although it is permitted) for variations such as complex substitutions or other events where all alleles have at least one base represented in their strings. If any of the `alt` alleles is a symbolic allele (an angle-bracketed ID String <ID>), then the padding base is required and `pos` denotes the coordinate of the base preceding the polymorphism. (String)
- `alt`: Alternate base(s). List of alternate non-reference alleles called on at least one of the samples. Options are strings made up of the bases A,C,G,T,N,*, (case insensitive) or an angle-bracketed ID String (<ID>) or a breakend replacement string as described in the VCFv4.2 section on breakends. The * allele is reserved to indicate that the allele is missing due to a upstream deletion. If there are no alternative alleles, the list should be empty. (String; no whitespace, commas, or angle-brackets are permitted in the ID String itself).

- `filter`: Filter status. A list of length 1 with value *PASS* if this position has passed all filters (if a call is made at this position). Otherwise, if the site has not passed all filters, a list of codes for filters that fail. [*q10*, *s50*] might indicate that at this site the quality is below 10 and the number of samples with data is below 50% of the total number of samples. *0* is reserved and should not be used as a filter string. If filters have not been applied, then the list should be empty. (List of strings with no white-space or semi-colons permitted).
- `format`: Genotype information (JSON-formatted). Key is 'GT'. Values are a list of strings with no white-space, semi-colons, commas or equals-signs.

```
{
  'chrom':<string>,
  'pos':<int>,
  'ref':<string>,
  'alt':[<string>, ...],
  'filter':[<string>, ...],
  'format': [VCFSampleFormatData_, ...]
}
```

Example VCFLine:

```
{
  'chrom':'13',
  'pos':32200123,
  'ref':T,
  'alt':['A'],
  'filter':[],
  'format': [
    {'sample-name':'human1-illumina', 'GT':'0/1'}
  ]
}
```

gVCFLine

Representation of gVCF line. Keys are the same as *VCFLine* with one additional key:

- `info`: Additional information (dictionary, optional) to indicate when a gVCF block ends. Only 1 valid key exists in this version: "END". Values are a list of integers. The list must be of length 1.

```
{
  'chrom':<string>,
  'pos':<int>,
  'ref':<string>,
  'alt':[<string>, ...],
  'filter':[<string>, ...],
  'format': [VCFSampleFormatData_, ...],
  'info':{"END":[<int>]}
}
```

Example gVCFLine:

```
{
  'chrom':'13',
  'pos':32199977,
  'ref':G,
  'alt':['<NON_REF>'],
  'filter':[],
  'format': [
    {'sample-name':'human1-illumina', 'GT':'0/0'}
  ],
  'info':{'END':[32200122]}
}
```

Another Valid gVCFLine Example:

```
  {
    'chrom':'13',
    'pos':32200123,
    'ref':T,
    'alt':['A','<NON_REF>'],
    'filter':[],
    'format': [
      {'sample-name':'human1-illumina', 'GT':'0/1'}
    ]
  }
```

7.2 Data Structures Specifications, v0.1.1

7.2.1 TilePosition

(No changes, see *TilePosition*.)

7.2.2 TilePositionRange

(No changes, see *TilePositionRange*.)

7.2.3 TileVariant

(No changes, see *TileVariant*.)

7.2.4 NotTileVariant

(No changes, see *NotTileVariant*.)

7.2.5 ClauseEntry

(No changes, see *ClauseEntry*.)

7.2.6 TileVariantClause

(No changes, see *TileVariantClause*.)

7.2.7 TileVariantLogic

(No changes, see *TileVariantLogic*)

7.2.8 TileVariantDetail

Todo

Specify md5sum for poorly sequenced tiles Different frequencies for more information - how many are well sequenced? How many are there, regardless of sequencing? Etc

The metadata information associated with one tile variant. Dictionary containing the keys:

- `tag-length`: The length of the tags.
- `start-tag`: The sequence of the start tag. Must be of length 0 (if the tile is at the start of the path) or the length specified by `tag-length`. Cannot have n's.
- `end-tag`: The sequence of the end tag. Must be of length 0 (if the tile is at the end of the path) or the length specified by `tag-length`. Cannot have n's.
- `is-start-of-path`: A boolean indicating whether the tile is at the start of the path.
- `is-end-of-path`: A boolean indicating whether the tile is at the end of the path.
- `sequence`: The sequence of the tile. May include n's.
- `md5sum`: TODO.
- `length`: The length of the tile sequence.
- `number-of-positions-spanned`: The number of tile positions this tile variant spans. Must be greater or equal to 1.
- `population-frequency`: The percentage of the population that contains this tile variant. Each phase counts as 1 entry in the population.
- `population-total`: The number of phases that contain a tile at this tile position. Each phase counts as 1 entry in the population.

```
{
    'tag-length': <int>,
    'start-tag': <string>,
    'end-tag': <string>,
    'is-start-of-path': <boolean>,
    'is-end-of-path': <boolean>,
    'sequence' : <string>,
    'md5sum': <string>,
    'length': <int>,
    'number-of-positions-spanned': <int>,
    'population-frequency': <float>,
    'population-total': <int>
}
```

Validations for each key are as follows:

```
TileVariantDetail['tag-length'] >= 1
TAG_LENGTH = TileVariantDetail['tag-length']
TileVariantDetail['start-tag'] matches '^[acgt]{TAG_LENGTH}$|^$'
TileVariantDetail['end-tag'] matches '^[acgt]{TAG_LENGTH}$|^$'
TileVariantDetail['sequence'] matches '[acgtn]'
TileVariantDetail['md5sum'] == MD5_hash_digest(TileVariantDetail['sequence'])
TileVariantDetail['length'] == len(TileVariantDetail['sequence'])
TileVariantDetail['number-of-positions-spanned'] >= 1
0 <= TileVariantDetail['population-frequency'] <= 1
TileVariantDetail['population-total'] >= 0
```

7.2.9 Locus

(No changes, see *Locus*)

7.2.10 CMPTuple

(No changes, see *CMPTuple*)

7.2.11 CMPFunction

(No changes, see *CMPFunction*)

7.2.12 VCF Data Structures Specifications

VCFInfoField

JSON-formatted information field format for a VCF or gVCF file. Keys that are not defined here may be included. Required keys are:

- ‘ID’, the name of the info field.
- ‘Number’, indicates the number of values that can be included with the INFO field. If the field has one value per alternate allele, the value should be ‘A’. If the field has one value for each possible allele (including the reference’), the value should be ‘R’. If the field has one value for each possible genotype, the value should be ‘G’. If the number of possible values varies, is unknown, or is unbounded, the value should be ‘.’.
- ‘Type’, options are ‘Integer’, ‘Float’, ‘Flag’, ‘Character’, and ‘String’.
- ‘Description’, the description of the info field.
- [optional] ‘Source’
- [optional] ‘Version’

If the ‘Type’ is equal to ‘Flag’, the INFO field does not contain a Value entry and the number should be 0.

```
{
    'ID': <string>,
    'Number':<int>|'A'|'R'|'G'|'.',
    'Type':<string>,
    'Description':<string>,
    'Source':<string>,
    'Version':<string>
}
```

VCFFilterField

JSON-formatted filter field format for a VCF or gVCF file. Keys that are not defined here may be included. Required keys are ‘ID’ and ‘Description’.

```
{
  'ID': <string>,
  'Description':<string>
}
```

VCFFormatField

JSON-formatted format field format for a VCF or gVCF file. Used for Genotype fields. Keys that are not defined here may be included. Required keys are:

- 'ID'
- 'Number', indicates the number of values that can be included with the INFO field. If the field has one value per alternate allele, the value should be 'A'. If the field has one value for each possible allele (including the reference'), the value should be 'R'. If the field has one value for each possible genotype, the value should be 'G'. If the number of possible values varies, is unknown, or is unbounded, the value should be '.'.
- 'Type', options are 'Integer', 'Float', 'Character', and 'String'
- 'Description'

```
{
  'ID': <string>,
  'Number':<int>|'A'|'R'|'G'|'.',
  'Type':<string>,
  'Description':<string>
}
```

VCFAlternativeAlleleField

JSON-formatted alternative allele field format for a VCF or gVCF file. Used to build symbols for alternate alleles (used for imprecise structural variants). Keys that are not defined here may be included. Required keys are 'ID' and 'Description'. The ID field indicates the type of structural variant and can be a colon-separated list of types and subtypes. The ID values are case sensitive and may not contain whitespace or angle brackets. The first level type must be one of the following:

- DEL, deletion relative to the reference
- INS, insertion of novel sequence relative to the reference
- DUP, region of elevated copy number relative to the reference
- INV, inversion of reference sequence
- CNV, copy number variable region (may be both deletion and duplication; should not be used when a more specific category may be applied)

Reserved subtypes include:

- DUP:TANDEM, tandem duplication
- DEL:ME, deletion of a mobile element relative to the reference
- INS:ME, insertion of a mobile element relative to the reference

```
{
  'ID': <string>,
  'Description':<string>
}
```

VCFContigField

JSON-formatted contig field format for a VCF or gVCF file. Keys that are not defined here may be included. Required keys are 'ID' and 'Description', and 'URL'. 'URL' points to the location of the contig.

```
{
  'ID': <string>,
  'Description':<string>,
  'URL':<string>
}
```

VCFSampleField

JSON-formatted sample field format for a VCF or gVCF file. Used to define sample to genome mappings. This is the only info given by the VCFv4.2 specifications. I believe the length of 'Genomes', 'Mixture', and 'Description' must be the same.

```
{
  'ID': <string>,
  'Genomes':[<string>, ...],
  'Mixture':[<string>, ...],
  'Description':[<string>, ...]
}
```

VCFMetaData

Representation of a VCF Header. JSON-formatted with keys:

- `fileformat`, whose value indicates the format of the VCFLine's returned by the Lightning server
- `fileDate`, the current date, format (YYYYMMDD)
- `source`, the Lightning server producing the VCF MetaData
- `reference`, the location of the reference fasta file used to generate the VCFLines
- `assembly`, same as `reference`
- [optional] `info`, for information field formats
- [optional] `filter`, for filters that have been applied to the data
- [optional] `format`, for genotype fields
- [optional] `alt`, for symbolizing imprecise structural variants
- [optional] `contig`, for pointing to sequence contigs
- [optional] `sample`, for defining sample to genome mappings
- [optional] `pedigree`, for defining relationships between genomes. Can be a list of name:genome or a url pointing to a pedigree database

```
{
  'fileformat':<string>,
  'fileDate':<int>,
  'source':<string>,
  'reference':<string>,
  'assembly':<string>,
  'info': [VCFInfoField_, ...],
  'filter': [VCFFilterField_, ...],
  'format': [VCFFormatField_, ...],
  'alt': [VCFAlternativeAlleleField_, ...],
  'contig': [VCFContigField_, ...],
  'sample': [VCFSampleField_, ...],
  'pedigree': [{<string>:<string>}, ...] OR <string>
}
```

Example:

```
{
  'fileformat':'VCFv4.2',
  'fileDate':20150921,
  'source':'Lightningv0.1.0',
  'reference':'1adbd1bd00358fe6ff2303ec8f3169ce+83454',
  'assembly':'1adbd1bd00358fe6ff2303ec8f3169ce+83454'
}
```

VCFSampleFormatData

JSON-formatted format field entry for a VCF or gVCF line. Used for Genotype fields of specific samples (identified by the `sample-name` key below). Further keys are the values associated with the 'ID' of *VCFFormatField*'s returned by *VCFMetaData* ['format']. Values are a list of strings with no white-space, semi-colons, commas or equals-signs. Useful predefined keys include:

- GT : genotype. The values associated with this key are strings of allele values for the specified sample in `sample-name`. The alleles are separated by / (for unphased) or | (for phased). The allele values are 0 for the reference allele (which is in the `ref` field of the *VCFLine*), 1 for the first allele listed in the `alt` field of the *VCFLine*, 2 for the second allele in the `alt` field of the *VCFLine* and so on. If a call cannot be made for a sample at a given locus, . should be specified for each missing allele in the GT field.
- PS : phase set. The values associated with this key are non-negative 32-bit integers indicating the phase set this genotype belongs to. A phase set is a set of phased genotypes. Phased genotypes for an individual sample that are on the same chromosome and have the same PS value are in the same phased set. A phase set specifies multi-marker haplotypes for the phased genotypes in the set. All phased genotypes that do not contain a PS subfield are assumed to belong to the same phased set. If the genotype in the GT field is unphased, the corresponding PS field is ignored.

One additional key (not defined in *VCFMetaData* ['format']) is required:

- `sample-name`, the name of the sample, which is normally defined by the column header in a VCF file.

```
{
  'sample-name': <string>,
  <string>:[<string>, ...],
  <string>:[<string>, ...],
  ...
}
```

VCFLine

Representation of VCF line. The info field maps the IDs from *VCFInfoField*'s defined in *VCFMetaData*. Required keys:

- `chrom`: Chromosome. An identifier from the reference genome or an angle-bracketed ID String (<ID>) pointing to a contig in the assembly file (the file pointed to by `assembly` in *VCFMetaData*). The colon symbol (:) must be absent from all chromosome names to avoid parsing errors when dealing with breakends. (String, no white-space permitted).

- `pos`: Position. The reference position, 1-indexed. Telomeres are indicated by using positions 0 or N+1, where N is the length of the corresponding chromosome or contig. (Integer).
- `id`: Identifier. List of unique identifiers if available. If there is no identifier available, then the list is empty. (List of strings, no white-space or semi-colons permitted
- `ref`: Reference base(s). Each base must be one of A,C,G,T,N (case insensitive). Multiple bases are permitted. The value matching the `pos` key refers to the position of the first base in this string. For simple insertions and deletions in which either the `ref` or one of the `alt` alleles would otherwise be null/empty, the `ref` and `alt` values must include the base before the event (which must be reflected in the `pos` field), unless the event occurs at position 1 on the contig in which case it must include the base after the event; this padding base is not required (although it is permitted) for variations such as complex substitutions or other events where all alleles have at least one base represented in their strings. If any of the `alt` alleles is a symbolic allele (an angle-bracketed ID String <ID>), then the padding base is required and `pos` denotes the coordinate of the base preceding the polymorphism. (String)
- `alt`: Alternate base(s). List of alternate non-reference alleles called on at least one of the samples. Options are strings made up of the bases A,C,G,T,N,*, (case insensitive) or an angle-bracketed ID String (<ID>) or a breakend replacement string as described in the VCFv4.2 section on breakends. The * allele is reserved to indicate that the allele is missing due to a upstream deletion. If there are no alternative alleles, the list should be empty. (String; no whitespace, commas, or angle-brackets are permitted in the ID String itself).
- `qual`: Quality. Phred-scaled quality score for the assertion made in ALT. -10*log10(prob[call in alt is wrong]). If `alt` is empty, then this is -10*log10(prob[variant]), otherwise, this is -10*log10(prob[no variant]). If this is unknown, return None.
- `filter`: Filter status. A list of length 1 with value *PASS* if this position has passed all filters (if a call is made at this position). Otherwise, if the site has not passed all filters, a list of codes for filters that fail. [*q10*, *s50*] might indicate that at this site the quality is below 10 and the number of samples with data is below 50% of the total number of samples. *0* is reserved and should not be used as a filter string. If filters have not been applied, then the list should be empty. (List of strings with no white-space or semi-colons permitted)
- `info`: Additional information (JSON-formatted). Keys are the values associated with the 'ID' key in the *VCFInfoField*'s of VCFMetaData_['info']. Values are a list of strings with no white-space, semi-colons, commas or equals-signs. List may be empty for Flag info keys.
- [optional] `format`: Genotype information (List of *VCFSampleFormatData*). Given if 'format' is defined in *VCFMetaData*.

```
{
  'chrom':<string>,
  'pos':<int>,
  'id':[<string>, ...],
  'ref':<string>,
  'alt':[<string>, ...],
  'qual':<float>|None,
  'filter':[<string>, ...],
  'info': {
    <string>:[<string>, ...],
    <string>:[<string>, ...],
    ...
  },
  'format': [VCFSampleFormatData, ...]
}
```

gVCFBlock

Genotype quality ranges used for banding. List of integers (length greater than or equal to 2). Each pair of integers creates a minGQ (inclusive) and maxGQ (exclusive) pair for a gVCFBlock. Must be strictly increasing, the first entry must be 0, and the last entry must be 2147483647 (the largest unsigned integer representable in 32 bits).

```
[0, <int>, ..., 2147483647]
```

gVCFMetaData

Representation of a gVCF Header. JSON-formatted with keys:

- `fileformat`, whose value indicates the format of the VCFLine's returned by the Lightning server
- `fileDate`, the current date, format (YYYYMMDD)
- `source`, the Lightning server producing the VCF MetaData
- `reference`, the location of the reference fasta file used to generate the VCFLines
- `assembly`, same as `reference`
- `alt`, for symbolizing imprecise structural variants. Includes 'NOT_REF' for the non-reference alternate allele.
- `info`, for information field formats. Includes 'END', which is used to indicate the end of a VCFLine.
- [optional] `gvcfblock`, for information about splitting gVCF blocks
- [optional] `filter`, for filters that have been applied to the data
- [optional] `format`, for genotype fields
- [optional] `contig`, for pointing to sequence contigs
- [optional] `sample`, for defining sample to genome mappings
- [optional] `pedigree`, for defining relationships between genomes. Can be a list of name:genome or a url pointing to a pedigree database

```
{
  'fileformat':<string>,
  'fileDate':<int>,
  'source':<string>,
  'reference':<string>,
  'assembly':<string>,
  'info': [
    {
      'ID': 'END',
      'Number':1,
      'Type':Integer,
      'Description':'Stop position of the interval'
    },
    VCFInfoField_,
    ...
  ],
  'alt': [
    {
      'ID': 'NOT_REF',
      'Description':'Represents any possible alternative allele at this location'
    },
    VCFAlternativeAlleleField_,
    ...
  ],
  'gvcfblock': gVCFBlock_,'
  'filter': [VCFFilterField_, ...],
  'format': [VCFFormatField_, ...],
  'contig': [VCFContigField_, ...],
  'sample': [VCFSampleField_, ...],
  'pedigree': [{<string>:<string>}, ...] OR <string>
}
```

CHAPTER

EIGHT

LIGHTNING API SPECIFICATIONS

We highly recommend reading the appropriate version of Data Structures Specifications before reading the API specifications you are interested in.

Components for each API specification:

- Server Namespace
- API Calls
- API Examples

Contents:

8.1 Lightning v0.1.0 API Specifications

We highly recommend reading Data Structures Specifications, v0.1.0 before diving into this API. These API only support well-sequenced tile variants. Note that RESTful API GET queries do not allow request data.

8.1.1 Lightning Server Namespace

```
/status : returns the API version running on the server
/tile-library
  /tag-sets : returns the tag set version information for all versions
              supported by this Lighting server instance.
    /{tag-set-identifier} : given the tag set version identifier, returns
                            information about that tag set.
      /paths : given the tag set version integer, returns the paths in that
               tag set.
        /{path-int} : given the tag set version identifier and the path integer,
                      returns information about that path.
      /tile-positions : given the tag set version identifier, returns the tile
                        positions in that tag set.
        /{tile-position-id} : given the tag set version identifier and tile
                              position identifier, returns information about
                              that tile position.
          /locus : given the tag set version identifier, tile position identifier,
                   and optional query parameters containing assembly information,
                   returns locus information about the tile position.
      /tile-variants : given a tag set version identifier, returns the tile
                       variants in that tag set in this Lightning server instance.
        /{tile-variant-id} : given the tag set version identifier and tile variant
                             identifier, returns details about the tile variant.
          /locus : given the tag set version identifier, tile variant identifier,
                   and optional query parameters containing assembly information,
                   returns locus information about the tile variant.
          /subsequence: given the tag set version identifier, tile variant
                        identifier, and query parameters containing locus
                        information, returns the subsequence of the tile variant
          /annotations: given the tag set version identifier and tile variant
                        identifier, returns the annotation identifiers applying
                        to that tile variant.
/annotations : returns a list of annotation identifiers loaded into the Lightning
               instance.
  /{annotation-id} : given an annotation id, returns the tile variants associated
                     with that annotation.
/callsets : returns a list of all genome names, termed callsets, loaded into this
            Lightning server instance.
  /{callset-name} : given the callset name, returns details about the callset.
    /gvcf : given the callset name and locus query parameters, returns a list
            of gVCF lines.
    /vcf : given the callset name and locus query parameters, returns a list of
           VCF lines.
    /tile-variants : given the callset name and tile position query parameters,
                     returns the tile variants the callset has at the given tile
                     position.
/assemblies : returns the available assemblies
  /{assembly-id} : returns the details about the assembly, including a list of
                   loaded loci (valid locations) on the Lightning server instance.
/searches : returns a list of searches that have been performed
  /{search-id} : returns the specific search and the answer of the search
```

8.1.2 GET /status

Request used to obtain the status of the Lightning server instance, which currently only includes the API version the server is running. Does not require any query parameters.

Response body:

```
{
  'api-version': <int>.<int>.<int>
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/status
```

Example response:

```
{ 'api-version' : 0.1.0 }
```

8.1.3 GET /tile-library/tag-sets

Request used to get the available tag sets on this Lightning instance. Does not require any query parameters. Returns a list of tag set unique identifiers (portable data hashes of the collection containing the tag set). This collection contains information about the tag set and the path dividers.

Response body:

```
[<string>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets
```

Example response:

```
['d87075c41962489cb9ce7d63da1d7841', '047ae54fba97385716acd2c552fae763']
```

8.1.4 GET /tile-library/tag-sets/{tag-set-identifier}

Request used to get information about the given tag set identifier. Does not require any query parameters. Provides the short integer identifier used by this server to represent the tag set (keyed by 'tag-set-integer').

Response body:

```
{
  'tag-set-identifier' : <string>,
  'tag-set-integer': <int> (base 16)
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response:

```
{
  'tag-set-identifier' : 'd87075c41962489cb9ce7d63da1d7841',
  'tag-set-integer' : 00
}
```

8.1.5 GET /tile-library/tag-sets/{tag-set-identifier}/paths

Request used to get the available paths for a specific tag set on this Lightning server instance. Tag set is specified using the tag set identifier in the uri. Does not require any query parameters. Returns a list of path integers (in base 16).

Response body:

```
[<int>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response for a BRCA Lightning Instance (a lightning server with 2 paths defined - 247 and 2c5):

```
[247, 2c5]
```

8.1.6 GET /tile-library/tag-sets/{tag-set-identifier}/paths/{path-int}

Request used to get information about a specific path for a specific tag set on this Lightning server instance. Tag set is specified using the tag set identifier in the uri. The path is identified using the path integer, written in base 16 in the uri. Does not require any query parameters. Returns a dictionary with path information for the specified path and tag set.

The number of tile positions for this path is provided under the 'num-positions' key.

Response body:

```
{
    'path' : <int> (base 16),
    'num-positions' : <int> (base 10)
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response:

```
{
    'path' : 2c5,
    'num-positions' : 12462
}
```

8.1.7 GET /tile-library/tag-sets/{tag-set-identifier}/tile-positions

Request used to get the available tile positions for a specific tag set on this Lightning server instance. Tag set is specified using the tag set identifier in the uri. Does not require any query parameters. In the future, might support query parameters filtering on information about the tile (like its path). Returns a list of tile position identifiers (*TilePosition*).

Response body:

```
[<TilePosition>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response for a BRCA Lightning Instance:

```
['00.247.0000', '00.247.0001', ..., '00.247.1bfb', '00.2c5.0000', ..., '00.2c5.30ae']
```

8.1.8 GET /tile-library/tag-sets/{tag-set-identifier}/tile-positions/{tile-position-id}

Request used to get information about a specific tile position for a specific tag set on this Lightning server instance. The tag set is specified using the tag set identifier in the uri. The tile position is identified using the *TilePosition* in the uri. Does not require any query parameters. Returns a dictionary with tile position information for the specified tag set and tile position.

Response body:

```
{
    'tile-position': <TilePosition>,
    'total-tile-variants': <int>, (base 10)
    'well-sequenced-tile-variants': <int>, (base 10)
    'num-genomes': <int> (base 10)
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example Response:

```
{
    'tile-position': '00.247.0000',
    'total-tile-variants': 25,
    'well-sequenced-tile-variants': 0,
    'num-genomes': 680
}
```

8.1.9 GET /tile-library/tag-sets/{tag-set-identifier}/tile-positions/{tile-position-id}/locus

Request used to get locus information about a specific tile position for a specific tag set on this Lightning server instance. The tag set is specified using the tag set identifier in the uri. The tile position is identified using the *TilePosition* in the uri. Does not require any query parameters, but an assembly identifier may be used to get information about a specific assembly. Returns a list of *Locus*'s. If no query parameters are specified, the list returned contains the loci for all assemblies in the Lightning server instance.

GET Query Parameters:

Parameter name	Type	Notes
`assembly-name`	<string>	Optional
`assembly-pdh`	<string>	Optional

Response body:

```
[<Locus>, ...]
```

Example Query Parameters:

Parameter name	Value
`assembly-name`	'hg19'
`assembly-pdh`	'dad94936d4144f5e0a289244d8be93e9+5735'

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response body:

```
[
  {
    'assembly-name': 'hg19',
    'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735',
    'chromosome-name': '13',
    'indexing': 0,
    'start-position': 32199976,
    'end-position': 32200225
  }
]
```

8.1.10 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants

Request used to get the available tile variants for a specific tag set on this Lightning server instance. Tag set is specified using the tag set identifier in the uri. Does not require any query parameters. In the future, might support query parameters filtering on information about the tile variant (like its path). Returns a list of tile variant identifiers (*TileVariant*).

Response body:

```
[<TileVariant>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response for a BRCA Lightning Instance:

```
[
  '00.247.0000.830003ac103a97d8f7992e09594ac68e',
  '00.247.0000.455577ff6b0d38188477ee2bfb2f0ea8',
  ...,
  '00.247.1bfb.c95325c08a449529143776e18561db71',
  '00.2c5.0000.1948117b4a56e4ad73d36dce185110fd',
  ...,
  '00.2c5.30ae.bc952f709d7419f7e103daa2b7e469a9'
]
```

8.1.11 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}

Request used to get information about a specific tile variant for a specific tag set on this Lightning server instance. The tag set is specified using the tag set identifier in the uri. The tile variant is identified using the *TileVariant* in the uri. Does not require any query parameters. Returns details about the specified *TileVariant* as a *TileVariantDetail*.

Response body:

```
<TileVariantDetail>
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response:

```
{
    'tile-variant':'00.2c5.30ae.bc952f709d7419f7e103daa2b7e469a9',
    'tag-length': 24,
    'start-tag': 'gccaaggagttttaaaactactga',
    'end-tag': '',
    'is-start-of-path': False,
    'is-end-of-path': True,
    'sequence' : 'gccaaggagttttaaaactactgatgcccacctcccacacccaaaagtctgattaattgatctagggtatggcctgag
    'md5sum': 'bc952f709d7419f7e103daa2b7e469a9',
    'length': 394,
    'number-of-positions-spanned': 1,
    'population-frequency': 0.5,
    'population-count': 150,
    'population-total': 300
}
```

8.1.12 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}/locus

Request used to get locus information about a specific tile variant for a specific tag set on this Lightning server instance. The tag set is specified using the tag set identifier in the uri. The tile variant is identified using the *TileVariant* in the uri. Does not require any query parameters, but an assembly identifier may be used to get information about a specific assembly. Returns a list of *Locus*'s. If no query parameters are specified, the list returned contains the loci for all assemblies in the Lightning server instance.

GET Query Parameters:

Parameter name	Type	Notes
`assembly-name`	<string>	Optional
`assembly-pdh`	<string>	Optional

Response body:

```
[<Locus>, ...]
```

Example Query Parameters:

Parameter name	Value
`assembly-name`	'hg19'
`assembly-pdh`	'dad94936d4144f5e0a289244d8be93e9+5735'

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response body:

```
[
  {
    'assembly-name': 'hg19',
    'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735',
    'chromosome-name': '13',
    'indexing': 0,
    'start-position': 32199976,
    'end-position': 32200225
  }
]
```

8.1.13 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}/subsequence

Request used to get a subsequence of a specific tile variant for a specific tag set on this Lightning server instance. The tag set is specified using the tag set identifier in the uri. The tile variant is identified using the *TileVariant* in the uri. Query parameters follow the *Locus* data structure. These query parameters are required. The response is a dictionary with one key: `sequence`.

If the *Locus* provided touches a locus outside of the tile variant, the API should return an error.

GET Query Parameters:

Parameter name	Type
`assembly-name`	<string>
`assembly-pdh`	<string>
`chromosome-name`	<string>
`indexing`	<int>
`start-position`	<int>
`end-position`	<int>

Response body:

```
{
  'sequence': <string>
}
```

Example Query Parameters:

Parameter name	Value
`assembly-name`	'hg19'
`assembly-pdh`	'dad94936d4144f5e0a289244d8be93e9+5735'
`chromosome-name`	'13'
`indexing`	0
`start-position`	32199976
`end-position`	32199983

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response body:

```
{
  'sequence': 'gggtac'
}
```

8.1.14 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}/annotations

Request used to get the annotations associated with a specific tile variant for a specific tag set on this Lightning server instance. The tag set is specified using the tag set identifier in the uri. The tile variant is identified using the *TileVariant* in the uri. Does not require any query parameters. The response is a list of annotation identifiers. If no annotation identifiers exist that are associated with that tile variant, returns an empty list.

Response body:

```
[ <string>, <string>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/tile-library/tag-sets/d87075c41962489cb9
```

Example response body:

```
[ 'annotation1', 'annotation3']
```

8.1.15 GET /annotiles

Request used to get all the annotations loaded into this Lightning server instance. Does not require any query parameters.

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/annotiles
```

Response body:

```
[ 'annotation0', 'annotation1', 'annotation2', 'annotation3']
```

8.1.16 GET /annotiles/{annotation-id}

Request used to determine which tile variants (*TileVariant*) are associated with a particular annotation identifier. Does not require any query parameters.

Response body:

```
[<TileVariant>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/annotiles/annotation0
```

Example response body:

```
['00.2c5.30ae.bc952f709d7419f7e103daa2b7e469a9']
```

8.1.17 GET /callsets

Request used to determine which callsets (called genomes) are loaded into this Lightning server instance. Returns a list of the names of the callsets. These names are expected to be identical to the names used in the phenotype database the user chooses. Does not require any query parameters.

Response body:

```
[<string>, <string>, ... ]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets
```

Example response body:

```
['human1-complete-genomics', 'human1-illumina', 'human2', 'assembly-hg19' ]
```

8.1.18 GET /callsets/{callset-name}

Request used to find details about a specific callset. Returns dictionary containing details about the callset. Phenotypic details may be passed by querying the phenotype database specified by the user, but this phenotype database is not part of Lightning.

Response body:

```
{
    'callset-name' : <string>,
    'callset-locator': <string>
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets/assembly-hg19
```

Example response body:

```
Response body: {
    'callset-name': 'assembly-hg19',
    'callset-locator': '1cf491c1ea99543da01c5a8f6b8a6dba+228008/hg19'
}
```

8.1.19 GET /callsets/{callset-name}/gvcf-header

Request used to obtain a valid gVCF header. Currently should not change based on the callset name specified in the uri. Requires query parameters specifying an assembly and an optional parameter specifying the gVCF blocks (*gVCFBlock*) to use. Responds with dictionary of type *gVCFMetaData*.

GET Query Parameters:

Parameter name	Type	Notes
`assembly-name`	<string>	
`assembly-pdh`	<string>	
`gvcf-block`	<list of ints>	Optional

Response body:

```
<gVCFMetaData>
```

Example Query Parameters:

Parameter name	Value
`assembly-name`	‘hg19’
`assembly-pdh`	‘dad94936d4144f5e0a289244d8be93e9+5735’

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets/human1-illumina/gvcf-header?ass
```

Example response body:

```
{
    'fileformat':'VCFc4.2',
    'fileDate':20150928,
    'source':'Lightningv0.1.0',
    'assembly':'dad94936d4144f5e0a289244d8be93e9+5735/hg19',
    'info': [
      {
        'ID': 'END',
        'Number':1,
        'Type':Integer,
        'Description':'Stop position of the interval'
      }
    ],
    'format': [
      {
        'ID': 'GT',
        'Number':1,
        'Type':'String',
        'Description':'Genotype'
      }
    ],
    'alt': [
      {
        'ID': 'NOT_REF',
        'Description':'Represents any possible alternative allele at this location'
      }
    ],
    'gvcfblock': [0, 2147483647]
}
```

8.1.20 GET /callsets/{callset-name}/gvcf

Request used to obtain a list of valid gVCF lines for specified callset in the uri. Requires query parameters specifying a *Locus* to retrieve lines for. The query parameters may include an optional parameter to specify the gVCF blocks (*gVCFBlock*) to use. Responds with a list of dictionaries of type *gVCFLine*. These represent the gVCF lines of the specified callset.

GET Query Parameters: ::

Parameter name	Type	Notes
`assembly-name`	<string>	
`assembly-pdh`	<string>	
`chromosome-name`	<string>	
`indexing`	<int>	
`start-position`	<int>	
`end-position`	<int>	
`gvcf-block`	<list of ints>	Optional

Response body:

```
[<gVCFLine>, ...]
```

Example query parameters:

Parameter name	Value
`assembly-name`	‘hg19’
`assembly-pdh`	‘dad94936d4144f5e0a289244d8be93e9+5735’
`chromosome-name`	‘13’
`indexing`	0
`start-position`	32199976
`end-position`	32200225

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets/human1-illumina/gvcf?assembly-n
```

Example response body:

```
[
  {
    'chrom':'13',
    'pos':32199977,
    'ref':G,
    'alt':['<NON_REF>'],
    'filter':[],
    'format': [
      {'sample-name':'human1-illumina', 'GT':'0/0'}
    ],
    'info':{'END':[32200122]}
  },
  {
    'chrom':'13',
    'pos':32200123,
    'ref':T,
    'alt':['A','<NON_REF>'],
    'filter':[],
    'format': [
      {'sample-name':'human1-illumina', 'GT':'0/1'}
    ]
  },
  {
    'chrom':'13',
    'pos':32200124,
    'ref':G,
    'alt':['<NON_REF>'],
    'filter':[],
    'format': [
      {'sample-name':'human1-illumina', 'GT':'0/0'}
    ],
    'info':{'END':[32200225]}
  }
]
```

8.1.21 GET /callsets/{callset-name}/vcf-header

Request used to obtain a valid VCF header. Currently should not change based on the callset name specified in the uri. Requires query parameters specifying an assembly. Responds with dictionary of type *VCFMetaData*.

GET Query Parameters:

Parameter name	Type
`assembly-name`	<string>
`assembly-pdh`	<string>

Response body:

```
<VCFMetaData>
```

Example Query Parameters:

Parameter name	Value
`assembly-name`	‘hg19’
`assembly-pdh`	‘dad94936d4144f5e0a289244d8be93e9+5735’

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets/human1-illumina/vcf-header?asse
```

Example response body:

```
{
    'fileformat':'VCFc4.2',
    'fileDate':20150928,
    'source':'Lightningv0.1.0',
    'assembly':'dad94936d4144f5e0a289244d8be93e9+5735/hg19',
    'format': [
      {
        'ID': 'GT',
        'Number':1,
        'Type':'String',
        'Description':'Genotype'
      }
    ]
}
```

8.1.22 GET /callsets/{callset-name}/vcf

Request used to obtain a list of valid VCF lines for specified callset in the uri. Requires query parameters specifying a *Locus* to retrieve lines for. Responds with a list of dictionaries of type *gVCFLine*. These represent the VCF lines of the specified callset.

GET Query Parameters: ::

Parameter name	Type
`assembly-name`	<string>
`assembly-pdh`	<string>
`chromosome-name`	<string>
`indexing`	<int>
`start-position`	<int>
`end-position`	<int>

Response body:

```
[<VCFLine>, ...]
```

Example query parameters:

Parameter name	Value
`assembly-name`	‘hg19’
`assembly-pdh`	‘dad94936d4144f5e0a289244d8be93e9+5735’
`chromosome-name`	‘13’
`indexing`	0
`start-position`	32199976
`end-position`	32200225

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets/human1-illumina/gvcf?assembly-n
```

Example response body:

```
[
  {
    'chrom':'13',
    'pos':32200123,
    'ref':T,
    'alt':['A'],
    'filter':[],
    'format': [
      {'sample-name':'human1-illumina', 'GT':'0/1'}
    ]
  }
]
```

8.1.23 GET /callsets/{callset-name}/tile-variants

Request used to obtain a list of tile variants for each phase for the callset specified in the uri. A list of *TilePosition* or *TilePositionRange* may be specified using optional query parameters. If no query parameters are specified, all tile variants are returned. Responds with a dictionary with two keys: `callset-name` and `tile-variants`. The value associated with tile variants is a list of lists. Each list represents a phase and contains objects of type *TileVariant*. Tile variants are returned if they intersect at with the given positions, even if they span outside the tile positions given.

GET Query Parameters: ::

Parameter name	Type	Notes
`tile-positions`	<TilePosition>|<TilePositionRange>	Optional

Response body:

```
{
    'callset-name':<string>,
    'tile-variants': [
        [ <TileVariant>, ...],
        [ <TileVariant>, ...],
        ...
    ]
}
```

Example query parameters:

Parameter name	Type
`tile-positions`	‘00.247.0000-0003’

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/callsets/human1-illumina/tile-variants?t
```

Example response body:

```
{
    'callset-name':'human1-illumina',
    'tile-variants': [
        [
          '00.247.0000.830003ac103a97d8f7992e09594ac68e',
          '00.247.0001.a31fd29383d072a5ccf7027ec37df093',
          '00.247.0002.a42a3e835440e21dda2cfd65162e85f0'
        ],
        [
          '00.247.0000.455577ff6b0d38188477ee2bfb2f0ea8',
          '00.247.0001.30c792a4fc1f0bd88dcc10907e6f27e6'
        ],
    ]
}
```

8.1.24 GET /assemblies

Request used to obtain a list of assemblies (*Assembly*) available in this Lightning server instance. Does not require any query parameters.

Response body:

```
[ <Assembly>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/assemblies
```

Example response body:

```
[
    {
      'assembly-name': 'hg19',
      'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735'
    },
    {
      'assembly-name': 'GRCh38',
      'assembly-pdh': '047ae54fba97385716acd2c552fae763+5735'
    }
]
```

8.1.25 GET /assemblies/{assembly-id}

Request used to obtain details about a specified assembly, given by the portable data hash in the uri, available in this Lightning server instance. Does not require any query parameters. Details are returned using a list of *Locus*'s.

Response body:

```
[ <Locus>, ...]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/assemblies/dad94936d4144f5e0a289244d8be9
```

Example response body:

```
[
    {
      'assembly-name': 'hg19',
      'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735',
      'chromosome-name': '13',
      'indexing': 0,
      'start-position': 32199976,
      'end-position': 34000000
    },
    {
      'assembly-name': 'hg19',
      'assembly-pdh': 'dad94936d4144f5e0a289244d8be93e9+5735',
      'chromosome-name': '17',
      'indexing': 0,
      'start-position': 40899976,
      'end-position': 44900000
```

```
        },
]
```

8.1.26 GET /searches/help

Returns the filters this Lightning server instance supports and the options available for each filter. If the filter 'tile-variants' is available, options will be an empty list for that filter, since those may be obtained by `GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants`. However, the key 'tile-variants' is included for completeness. The filter 'callsets' is treated identically: options will be an empty list for that filter, the key is included for completeness, and the options for 'callsets' may be obtained by `GET /callsets`. Does not require any query parameters. Details are returned in a dictionary with the keys matching the available filters. The values matching these keys are the available options for those filters.

Response body:

```
{
    <string> : [ <string>, <string>, ... ],
    <string> : [ <string>, <string>, ... ],
    ...
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/searches/help
```

Example response body:

```
{
    'tile-variants' : [ ],
    'callsets': [ ],
    'phasing': ['any-phase', 'all-phases']
}
```

8.1.27 GET /searches

Returns the searches that have been performed on this Lightning server instance. Does not require any query parameters. Details are returned as a list of pipeline uuids (*ArvadosUUID*).

Response body:

```
[<ArvadosUUID>, ... ]
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/searches
```

Example response body:

```
['su92l-d1hrv-lcrzu0qrs1iiu03', 'su92l-d1hrv-vg62va2y75c5wxm']
```

8.1.28 GET /searches/{search-id}

Returns the details about the search, given by the Arvados UUID (*ArvadosUUID*) in the uri, that has been performed on this Lightning server instance. Does not require any query parameters. Details are returned as a dictionary. The value associated with 'response' will be None if the pipeline is still running or failed.

Response body:

```
{
  'search-id': <ArvadosUUID>,
  'parameters': {
    <string>:[<string>],
    ...
  },
  'response': None | [<string>, ...]
}
```

Example Query:

```
curl -H "Accept:application/json" http://localhost:8888/searches/su92l-d1hrv-lcrzu0qrs1iiu03
```

Example response body:

```
{
  'search-id': 'su92l-d1hrv-lcrzu0qrs1iiu03',
  'filters': {
      'tile-variants': [['00.247.0000.830003ac103a97d8f7992e09594ac68e']],
      'phasing': ['any-phase']
  },
  'response': ['human1-complete-genomics', 'human1-illumina', 'assembly-hg19']
}
```

8.1.29 POST /searches

Creates a search with the specified parameters. The parameters should be available in the response `GET /searches/help`. If the parameter 'tile-variants' is used, the value associated with that key should be of type (*TileVariantLogic*). Returns the pipeline uuid (format *ArvadosUUID*) of that search. The output of that pipeline will be a list of callsets matching the search.

Request body:

```
{
    <string> :  [ <string>, <string>, ...],
    ...
}
```

Response body:

```
<ArvadosUUID>
```

Example Query:

```
curl -i -X POST -H "Content-Type:application/json" http://localhost:8888/searches -d '{"tile-var
```

Example response body:

```
'su92l-d1hrv-lcrzu0qrs1iiu03'
```

8.2 Reasoning Behind the API

API design! Here are some explanations!

Todo

Document API explanations

8.3 Lightning Errors

8.3.1 UnknownAssembly

Used when user requests an assembly that is not loaded into the Lightning instance

8.4 Lightning v0.1.1 API Specifications

We highly recommend reading Data Structures Specifications, v0.1.1 before diving into this API. Note that RESTful API GET queries do not allow request data.

8.4.1 Lightning Server Namespace

```
/status : returns the API version running on the server
/tile-library
  /tag-sets : returns the tag set version information for all versions
              supported by this Lighting server instance.
    /{tag-set-identifier} : given the tag set version identifier, returns
                            information about that tag set.
      /paths : given the tag set version integer, returns the paths in that
               tag set.
        /{path-int} : given the tag set version identifier and the path integer,
                      returns information about that path.
      /tile-positions : given the tag set version identifier, returns the tile
                        positions in that tag set.
        /{tile-position-id} : given the tag set version identifier and tile
                              position identifier, returns information about
                              that tile position.
          /locus : given the tag set version identifier, tile position identifier,
                   and optional query parameters containing assembly information,
                   returns locus information about the tile position.
      /tile-variants : given a tag set version identifier, returns the tile
                       variants in that tag set in this Lightning server instance.
        /{tile-variant-id} : given the tag set version identifier and tile variant
                             identifier, returns details about the tile variant.
          /locus : given the tag set version identifier, tile variant identifier,
                   and optional query parameters containing assembly information,
                   returns locus information about the tile variant.
          /subsequence: given the tag set version identifier, tile variant
                        identifier, and query parameters containing locus
                        information, returns the subsequence of the tile variant
          /annotations: given the tag set version identifier and tile variant
                        identifier, returns the annotation identifiers applying
                        to that tile variant.
/annotations : returns a list of annotation identifiers loaded into the Lightning
               instance.
  /{annotation-id} : given an annotation id, returns the tile variants associated
                     with that annotation.
/callsets : returns a list of all genome names, termed callsets, loaded into this
```

```
            Lightning server instance.
  /{callset-name} : given the callset name, returns details about the callset.
    /gvcf : given the callset name and locus query parameters, returns a list
            of gVCF lines.
    /vcf : given the callset name and locus query parameters, returns a list of
           VCF lines.
    /tile-variants : given the callset name and tile position query parameters,
                     returns the tile variants the callset has at the given tile
                     position.
/assemblies : returns the available assemblies
  /{assembly-id} : returns the details about the assembly, including a list of
                   loaded loci (valid locations) on the Lightning server instance.
/searches : returns a list of searches that have been performed
  /{search-id} : returns the specific search and the answer of the search
```

8.4.2 GET /status

Returns the status of the server, including the api-version running on the server, possibly the current load level, etc. Does not require any query parameters.

Response body:

```
    {
      'api-version': <int>[.<int>]*,
      ...
    }
```

8.4.3 GET /tile-library/tag-sets

Probably will be unchanged from v0.1.0.

8.4.4 GET /tile-library/tag-sets/{tag-set-identifier}

Probably will be unchanged from v0.1.0.

8.4.5 GET /tile-library/tag-sets/{tag-set-identifier}/paths

Probably will be unchanged from v0.1.0.

8.4.6 GET /tile-library/tag-sets/{tag-set-identifier}/paths/{path-int}

Probably will be unchanged from v0.1.0.

8.4.7 GET /tile-library/tag-sets/{tag-set-identifier}/tile-positions

Might be unchanged from v0.1.0. Might support query parameters filtering on information about the tile (like its path).

8.4.8 GET /tile-library/tag-sets/{tag-set-identifier}/tile-positions/{tile-position-id}

Probably will be unchanged from v0.1.0.

8.4.9 GET /tile-library/tag-sets/{tag-set-identifier}/tile-positions/{tile-position-id}/locus

Probably will be unchanged from v0.1.0.

8.4.10 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants

Might be unchanged from v0.1.0. Might support query parameters filtering on information about the tile variant (like its path).

8.4.11 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}

Specifications will probably will be unchanged from v0.1.0. But the response will change because the data structure changes.

8.4.12 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}/locus

Probably will be unchanged from v0.1.0. Will want a batch request to view multiple tile variants at a time.

8.4.13 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}/subsequence

Probably will be unchanged from v0.1.0. Will want a batch request to view multiple tile variants at a time.

8.4.14 GET /tile-library/tag-sets/{tag-set-identifier}/tile-variants/{tile-variant-id}/annotations

Probably will be unchanged from v0.1.0. Will want a batch request to view multiple tile variants at a time.

8.4.15 GET /annotiles

Probably will be unchanged from v0.1.0.

8.4.16 GET /annotiles/{annotation-id}

Probably will be unchanged from v0.1.0.

8.4.17 GET /callsets

Probably will be unchanged from v0.1.0.

8.4.18 POST /callsets

Creates a callset (called genomes) in this Lightning server instance. Returns an Arvados UUID pointing to the pipeline instance adding the callset. The callset will appear on the server once the pipeline finishes successfully.

Request body:

```
{
  'callset-name': <string>,
  'callset-format': <string>,
  'callset-collection-pdh': <string>
}
```

Response body:

```
<ArvadosUUID>
```

8.4.19 GET /callsets/{callset-name}

Probably will be unchanged from v0.1.0. Unlikely to include phenotypic query options.

8.4.20 GET /callsets/{callset-name}/gvcf-header

Functionality will probably remain the same as v0.1.0. Underlying specifications will change with data structure.

8.4.21 GET /callsets/{callset-name}/gvcf

Functionality will probably remain the same as v0.1.0. Underlying specifications will change with data structure. Might want to implement a batch query for multiple loci. Being able to name the locus would also be helpful.

8.4.22 GET /callsets/{callset-name}/vcf-header

Functionality will probably remain the same as v0.1.0. Underlying specifications will change with data structure.

8.4.23 GET /callsets/{callset-name}/vcf

Functionality will probably remain the same as v0.1.0. Underlying specifications will change with data structure. Might want to implement a batch query for multiple loci. Being able to name the locus would also be helpful.

8.4.24 GET /callsets/{callset-name}/tile-variants

Might be unchanged from v0.1.0. Might want to include phase group information. Will probably want to implement a batch query for getting tile variants for multiple callsets at once.

8.4.25 GET /assemblies

Probably will be unchanged from v0.1.0.

8.4.26 GET /assemblies/{assembly-id}

Probably will be unchanged from v0.1.0.

8.4.27 GET /searches/help

Probably will be unchanged from v0.1.0.

8.4.28 GET /searches

Probably will be unchanged from v0.1.0.

8.4.29 GET /searches/{search-id}

Will be expanded to allow searching on both callsets and tile variants.

Want to be able to find callsets with specific tile variants.

Want to be able to find tile variants with specific qualities: REGEX on sequences, start tag, and/or end tag. Comparison on length, number of positions spanned, population frequency, population total. True/False check on whether the tile is at the start of a path or at the end of the path.

8.4.30 POST /searches

Probably will be unchanged from v0.1.0.

8.5 Batch Processing

The API calls are excellent for one-off queries. However, many users may wish to use these calls many times for many samples and/or over large parts of the genome. To speed up these queries and reduce load on the server, Lightning plans to support batch processing.

Todo

Document these better

8.5.1 Filter to Find Tile Variants Matching Filters

Finds tile variants fulfilling a specified filters

- REGEX on sequence, start tag, or end tag
- Float comparison on length, the number of positions spanned, the frequency of this tile, or the number of callsets called at this position
- True/False on whether the tile is at the start of path or at the end of the path

8.5.2 Find Loci of Multiple Tiles at Once

Todo

Document

8.5.3 Retrieve the Library Around a Group of Loci

Returns a list of tile-positions touching given loci, a dictionary of tile-variants touching those loci with the sequence of the tile variant, cut off if appropriate, and the tag-length. We should be able to name the loci for convenience.

Todo

Should document the INDEL behavior. Previous documentation is at lightning-dev4.curoverse.com/pad/p/lightning-indel-behavior

8.6 Versioning

URL Parameter Versioning: The client specifies the version as part of the URL path:

```
GET /v0.1.0/status HTTP/1.1
Host: lightning.curoverse.com
Accept: application/json
```

For more information and implementation details, reference http://www.django-rest-framework.org/api-guide/versioning/.

8.7 Paging

Limit Offset Pagination: The client specifies the limit and offset using request query parameters

`limit`: indicates the maximum number of items to return. It's not required. Default limit is 100. Maximum limit is 1000.

`offset`: indicates the starting position of the query in relation to the complete set of unpaginated items

For information and implementation details, reference http://www.django-rest-framework.org/api-guide/pagination/.

CHAPTER

NINE

SOFTWARE DEVELOPMENT KITS

This section is here for completeness: we currently do not have sdks available for Lightning.

CHAPTER

TEN

SPRITE

Sprite is a django web application that provides visualizations and easy interaction with Lightning, as well as preliminary phenotype and tile variant annotation databases.

Todo

- Specify phenotype database
- Specify annotation database(s)

10.1 Sprite Annotation Database

Need to include CAVA and ClinVar annotations.

10.1.1 CAVA

CAVA depends on multiple inputs. Luckily, Arvados stores all these inputs along with the output. Possible CAVA dependencies include the reference genome input used, the ensembl transcript database input used, the dbSNP database file used, and the definition of impact used for the CAVA pipeline. Each tile has a VCF generated using it, which CAVA is then run on.

This database will use the CSV output from the CAVA pipeline. This outputs a **CSV file** with the following columns:

Index	Name	Description
0	ID	i.e. Variant call ID taken from the input file
1	CHROM	i.e. chromosome of variant
2	POS	i.e. genomic position of variant
3	REF	i.e. reference allele of variant
4	ALT	i.e. alternative allele of variant
5	QUAL	i.e. QUAL value in the input VCF record
6	FILTER	i.e. FILTER value in the input VCF record
7	TYPE	i.e. value of TYPE annotation flag
8	ENST	i.e. value of ENST annotation flag
9	GENE	i.e. value of GENE annotation flag
10	TRINFO	i.e. value of TRINFO annotation flag
11	LOC	i.e. value of LOC annotation flag
12	CSN	i.e. value of CSN annotation flag
13	CLASS	i.e. value of CLASS annotation flag
14	SO	i.e. value of SO annotation flag
15	IMPACT	i.e. value of IMPACT annotation flag
16	ALTFLAG	i.e. value of ALTFLAG annotation flag
17	ALTCLASS	i.e. value of ALTCLASS annotation flag
18	ALTSO	i.e. value of ALTSO annotation flag
19	DBSNP	i.e. value of DBSNP annotation flag

The output will be saved in a **database** with the following columns per entry:

Type	Details of Type	Name	Description
string	ArvadosUUID	pipelineuuid	Pipeline UUID
string	‘,’-separated	vcfinfo	VCF info. 5 comma-separated values (ID, CHROM, POS, REF, ALT).Indices (0-based) of CSV output of CAVA are 0,1,2,3,4.
string		quality	QUAL; also VCF info but more likely to get queried on. Index (0-based) of CSV output of CAVA is 5.
string		filter	FILTER; also VCF info but more likely to get queried on. Index (0-based) of CSV output of CAVA is 6.
string		type	TYPE; includes SNP, INDEL, and COMPLEX. Index (0-based) of CSV output of CAVA is 7.
string		enst	ENST; Ensembl transcript identifier applying to this position in the tile variant. Index (0-based) of CSV output of CAVA is 8.
string		gene	GENE; Gene name - HGNC symbol. Index (0-based) of CSV output of CAVA is 9.
string	‘/’-separated	trinfo	TRINFO; Transcript info in a /-separated string including: strandedness, length of transcript, number of exons, and length of coding DNA. Index (0-based) of CSV output of CAVA is 10.
string		loc	LOC; Location of variant within the transcript, which includes exon, intron, UTR3/5 regions. Index (0-based) of CSV output of CAVA is 11.
string		csn	CSN; Clinical sequencing nomenclature. Index (0-based) of CSV output of CAVA is 12.
string	‘,’-separated	class	CLASS; Includes ESS, FS, SG, and NSY. Index (0-based) of CSV output of CAVA is 13.
string	‘,’-separated	so	SO; Includes stop_gained and frameshift_variant. Index (0-based) of CSV output of CAVA is 14.
integer		impact	IMPACT. Index (0-based) of CSV output of CAVA is 15.
string		altflag	ALTFLAG. Index (0-based) of CSV output of CAVA is 16.
string		altclass	ALTCLASS. Index (0-based) of CSV output of CAVA is 17.
string		altso	ALTSO. Index (0-based) of CSV output of CAVA is 18.
string		dbsnp	DBSNP. Index (0-based) of CSV output of CAVA is 19.

10.1.2 ClinVar

Type	Details of Type	Name	Description
string	Unknown	acc	Variant accession and versions; Keyed by CLNACC in ClinVar VCF file.
string	Unknown	clinical-significance	Variant Clinical Significance, 0 - Uncertain significance, 1 - not provided, 2 - Benign, 3 - Likely benign, 4 - Likely pathogenic, 5 - Pathogenic, 6 - drug response, 7 - histocompatibility, 255 - other. Keyed by CLNSIG in ClinVar VCF file.
string	Unknown	diseasedb-name	Variant disease database name. Keyed by CLNDSDB.
string	Unknown	disease-name	Variant disease name. Keyed by CLNDBN.
string	Unknown	hgvs	Variant names from HGVS. Keyed by CLNHGVS in ClinVar VCF file.
string	Unknown	review-status	ClinVar Review Status, mult - Classified by multiple submitters, single - Classified by single submitter, not - Not classified by submitter, exp - Reviewed by expert panel, prof - Reviewed by professional society; Keyed by CLNREVSTAT in ClinVar VCF file.

Keys in ClinVar VCF file that will be useful for assigning ClinVar info later:

- CLNALLE (integer): Variant alleles from REF or ALT columns. 0 is REF, 1 is the first ALT allele, etc. This is used to match alleles with other corresponding clinical (CLN) INFO tags. A value of -1 indicates that no allele was found to match a corresponding HGVS allele name.

Unused Keys:

- CLNORIGIN (string): Allele Origin. One or more of the following values may be added: 0 - unknown; 1 - germline; 2 - somatic; 4 - inherited; 8 - paternal; 16 - maternal; 32 - de-novo; 64 - biparental; 128 - uniparental; 256 - not-tested; 512 - tested-inconclusive; 1073741824 - other
- CLNSRC (string): Variant Clinical Channels
- CLNSRCID (string): Variant Clinical Channel IDs
- CLNDSDBID (string): Variant disease database ID

10.1.3 Ideas from Biodata for Further Annotations

```
protocol VariantAnnotations {
  enum Expression {UP, DOWN}

  record Xref {
    union { null, string } id;
    union { null, string } src;
  }

  record Score {
    union { null, double } score;
    union { null, string } source;
    union { null, string } description;
  }

  record ConsequenceTypeEntry {
    union { null, string } soName;
    union { null, string } soAccession;
  }

  record ExpressionValue {
    union { null, string } experimentalFactor;
    union { null, string } factorValue;
    union { null, string } experimentId;
    union { null, string } technologyPlatform;
    Expression expression;
    union { null, float } pvalue;
  }

  record ConsequenceType {
    union { null, string } geneName;
    union { null, string } ensemblGeneId;
    union { null, string } ensemblTranscriptId;
    union { null, string } strand;
    union { null, string } biotype;
    union { null, int } cDnaPosition;
    union { null, int } cdsPosition;
    union { null, int } aaPosition;
    union { null, string } aaChange;
    union { null, string } codon;
    union { null, string } functionalDescription;
    union { null, array<Score> } proteinSubstitutionScores;
    union { null, array<ConsequenceTypeEntry> } soTerms;
    union { null, array<ExpressionValue> } expressionValues;
  }

  record PopulationFrequency {
    union { null, string } study;
    union { null, string } pop;
    union { null, string } superPop;
    union { null, string } refAllele;
    union { null, string } altAllele;
    union { null, float } refAlleleFreq;
    union { null, float } altAlleleFreq;
    union { null, float } refHomGenotypeFreq;
    union { null, float } hetGenotypeFreq;
    union { null, float } altHomGenotypeFreq;
  }

  record CaddScore {
    union { null, string } transcriptId;
    union { null, float } cScore;
    union { null, float } rawScore;
  }

  record VariantAnnotation {
    union { null, string } chromosome;
    union { null, int } start;
    union { null, int } end;
    union { null, string } reference;
    union { null, string } alternate;
    union { null, string } id;
    union { null, array<Xref> } xrefs;
    union { null, array<string> } hgvs;
    union { null, array<ConsequenceType> } consequenceTypes;
```

```
    union { null, array<Score> } conservation;
    union { null, array<PopulationFrequency> } populationFrequencies;
    //union { null, array<CaddScore> } caddScore;
    union { null, map<array<string>> } geneDrugInteraction;
    union { null, map<string> } variantTraitAssociation;
    union { null, map<string> } additionalAttributes;
  }
}

protocol Variants {
  enum VariantType {
      SNP,
      SNV,
      MNP,
      MNV,
      INDEL,
      SV,
      CNV,
      NO_VARIATION,
      SYMBOLIC,
      MIXED}

  enum AllelesCode {
      ALLELES_OK,
      ALLELES_MISSING,
      MULTIPLE_ALTERNATES,
      HAPLOID
      }

  record VariantHardyWeinbergStats {
      union { null, float } chi2;
      union { null, float } pValue;
      union { null, int } n;
      union { null, int } n_AA_11;
      union { null, int } n_Aa_10;
      union { null, int } n_aa_00;
      union { null, float } e_AA_11;
      union { null, float } e_Aa_10;
      union { null, float } e_aa_00;
      union { null, float } p;
      union { null, float } q;
  }

  record VariantStats {
      union { null, string } refAllele;
      union { null, string } altAllele;
      union { null, int } refAlleleCount;
      union { null, int } altAlleleCount;
      union { null, int } missingAlleles;
      union { null, int } missingGenotypes;
      union { null, float } refAlleleFreq;
      union { null, float } altAlleleFreq;
      union { null, float } maf;
      union { null, float } mgf;
      union { null, string } mafAllele;
      union { null, string } mgfGenotype;
      union { null, boolean } passedFilters;
      union { null, int } mendelianErrors;
      union { null, float } casesPercentDominant;
      union { null, float } controlsPercentDominant;
      union { null, float } casesPercentRecessive;
      union { null, float } controlsPercentRecessive;
      union { null, float } quality;
      union { null, int } numSamples;
      VariantType variantType;
      VariantHardyWeinbergStats hw;
  }

  record VariantSourceEntry {
      union { null, string } studyId;
      union { null, string } fileId;
      union { null, array<string> } secondaryAlternates = null;
      string format;
      //map<map<string>> samplesData;
      array<array<string>> samplesData;
      map<VariantStats> stats;
      map<string> attributes;
  }

  record Genotype {
      string reference;
      string alternate;
      array<int> allelesIdx =[];
      boolean phased;
  }

  record Variant {
      string chromosome;
      int start;
      int end;
      string reference;
      string alternate;
      array<string> ids = [];
      int length;
      VariantType type;
      map<array<string>> hgvs;
      array<VariantSourceEntry> studies;
      union {null, VariantAnnotation} annotation;
  }

  record VariantFileMetadata {
      string fileId;
      string studyId;
      union { null, string } fileName = null;
      union { null, string } studyName = null;
      array<string> samples = [];
      map<string> metadata;
  }
}
```

Note: Please keep in mind Lightning is under development, as is its documentation. Feel free to file bugs or documentation errors at https://dev.arvados.org/projects/lightning.

Todo

Make sure https://dev.arvados.org/projects/lightning is public and a well-behaved redmine account.

For an introduction, basic information about using Lightning, and Lightning's design document, see Getting started.

For a description of the process of tiling (the abstraction of genomic sequences that makes Lightning possible) and the functions and pipelines we provide for tiling genomes, see Tiling Overview.

For a description of our representation of tiled genomes (Compact Genome Format), see Compact Genome File (CGF) Format.

For a description of our in-memory database for tiled genomes (Lantern), including the REST API it supports, see Lantern Specifications.

For a description of our in-memory database for tile variants (the tile library), see Tile Library.

For a description of Annotile, the way we support annotations of tile variants, how to import annotation software, and how to add human-generated annotations, see Annotile: Annotating Tile Variants.

For a description of Lightning Data Structures, used for interaction with the Lightning APIs, see Data Structures Specifications.

For a description of Lightning APIs, see Lightning API Specifications.

For a description of Lightning Software Development Kits, see Software Development Kits.

Finally, for a description of the web browser application that runs on Lightning (Sprite), see Sprite.

CHAPTER ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search