CGAP-Portal

Feb 21, 2023

Contents

Welcome to CGAP! We are a team of scientists, clinicians, and developers who aim to streamline the clinical genetics workflow. The following locations are different deployments of our data portal:

- cgap-dbmi DBMI production account
- cgap-training demo account for potential users
- cgap-devtest general development/testing account
- cgap-wolf workflow development account

Be warned that features are under active development and may not be stable! Visit the demo account for the best experience. To get started, read the following documentation on the infrastructure and how to work with the data model:

Infrastructure

CHAPTER 1

CGAP Infrastructure Landing Page

Welcome! This landing page contains some information on what CGAP is and how to get started with the deployment of CGAP.

1.1 Overview

The Computational Genome Analysis Platform (CGAP) is an intuitive, open-source analysis tool designed to support complex research & clinical genomics workflows. It implements powerful variant discovery & diagnostic tools for individual samples and cohorts with clinical accuracy and reporting capabilities in one platform. CGAP is built on the AWS Cloud and is intended to be used within a single isolated AWS Account.

If you are interested in deploying CGAP, access to the infrastructure repository (4dn-cloud-infra) can be negotiated. Once the repository is open source this step will not be necessary, but at this time the system is considered to be in alpha stage release only, available on a confidential basis to a limited audience for early feedback. After we have assessed the stability we will open source this repository under an appropriate license.

Until then, please reach out the CGAP Operations team for access to the code. We will work with you to deploy and adapt CGAP to the needs of your project.

CHAPTER 2

Local Installation

2.1 Docker Instructions

The portal is meant to be run via Docker, but running tests or a local deployment outside of Docker requires the legacy instructions below. For Docker instructions see docker local docs..

2.2 Legacy Instructions

The following instructions are for running a CGAP deployment with macOS and homebrew.

Note that as of summer 2021, these instructions are out of date. Please refer to the Docker setup. There are no guarantees the legacy instructions will work from this point forward.

CGAP is known to work with Python 3.7.x and 3.8.x and will not work with Python 3.9 or greater. If part of the HMS team, it is recommended to use a high patch version, such as Python 3.8.13, since that's what we try to do with our servers, but any version of 3.8 should work if you find you are unable to install that particular patch version. It is best practice to create a fresh Python virtualenv using one of these versions before proceeding to the following steps.

- Step 0: Obtain AWS keys. These will need to added to your environment variables or through the AWS CLI (installed later in this process).
- Step 1: Verify that homebrew is working properly:

```
$ brew doctor
```

- Step 2: Install or update dependencies:

```
$ brew install libevent libmagic libxml2 libxslt openssl postgresql␣
↪graphviz nginx python3
$ brew install freetype libjpeg libtiff littlecms webp  # Required by␣
↪Pillow
$ brew cask install homebrew/cask-versions/adoptopenjdk8
```

(continues on next page)

(continued from previous page)

```
$ brew tap homebrew/versions
$ brew install opensearch node@16
```

You may need to link the brew-installed opensearch:

```
$ brew link --force opensearch
```

If you are migrating from elasticsearch@6 to opensearch:

```
$ brew uninstall elasticsearch@6
$ brew install opensearch
```

Note that this may bring in a new JDK.

If you need to update dependencies:

```
$ brew update
$ brew upgrade
$ rm -rf encoded/eggs
```

- Step 3: Run make:

```
$ make build

NOTE:
If you have issues with postgres or the python interface to it␣
↪(psycogpg2) you
probably need to install postgresql via homebrew (as above)
If you have issues with Pillow you may need to install new xcode command␣
↪line tools:
- First update Xcode from AppStore (reboot)
$ xcode-select --install
```

If you wish to completely rebuild the application, or have updated dependencies: $ make clean

Then goto Step 3.

- Step 4: Start the application locally

In one terminal startup the database servers and nginx proxy with:

```
$ make deploy1
```

This will first clear any existing data in /tmp/encoded. Then postgres and elasticsearch servers will be initiated within /tmp/encoded. An nginx proxy running on port 8000 will be started. The servers are started, and finally the test set will be loaded.

In a second terminal, run the app with:

```
$ make deploy2
```

Indexing will then proceed in a background thread similar to the production setup.

Running the app with the *–reload* flag will cause the app to restart when changes to the Python source files are detected:

```
$ bin/pserve development.ini --reload
```

If doing this, it is highly recommended to set the following environment variable to override the default file monitor used. The default monitor on Unix systems is watchman, which can cause problems due too tracking too many files and degrade performance. Use the following environment variable:

```
$ HUPPER_DEFAULT_MONITOR=hupper.polling.PollingFileMonitor
```

Browse to the interface at http://localhost:8000/.

2.3 Running tests

To run specific tests locally:

```
$ bin/test -k test_name
```

To run with a debugger:

```
$ bin/test --pdb
```

Specific tests to run locally for schema changes:

```
$ bin/test -k test_load_workbook
$ bin/test -k test_edw_sync
```

Run the Pyramid tests with:

```
$ bin/test
```

Note: to run against chrome you should first:

```
$ brew install chromedriver
```

Run the Javascript tests with:

```
$ npm test
```

Or if you need to supply command line arguments:

```
$ ./node_modules/.bin/jest
```

2.4 Building Javascript

Our Javascript is written using ES6 and JSX, so needs to be compiled using babel and webpack.

To build production-ready bundles, do:

```
$ npm run build
```

(This is also done as part of running buildout.)

To build development bundles and continue updating them as you edit source files, run:

```
$ npm run dev-quick
```

The development bundles are not minified, to speed up building.

2.5 Notes on SASS/Compass

We use the SASS and node-sass CSS preprocessors. The buildout installs the SASS utilities and compiles the CSS. When changing the SCSS source files you must recompile the CSS using one of the following methods:

2.6 Compiling "on the fly"

Node-sass can watch for any changes made to .scss files and instantly compile them to .css. To start this, from the root of the project do:

```
$ npm run watch-scss
```

2.7 Force compiling

```
$ npm run build-scss
```

2.8 SublimeLinter

To setup SublimeLinter with Sublime Text 3, first install the linters:

```
$ easy_install-2.7 flake8
$ npm install -g jshint
$ npm install -g jsxhint
```

After first setting up Package Control (follow install and usage instructions on site), use it to install the following packages in Sublime Text 3:

- sublimelinter
- sublimelinter-flake8
- sublimelinter-jsxhint
- jsx
- sublimelinter-jshint

CHAPTER 3

Infrastructure Overview

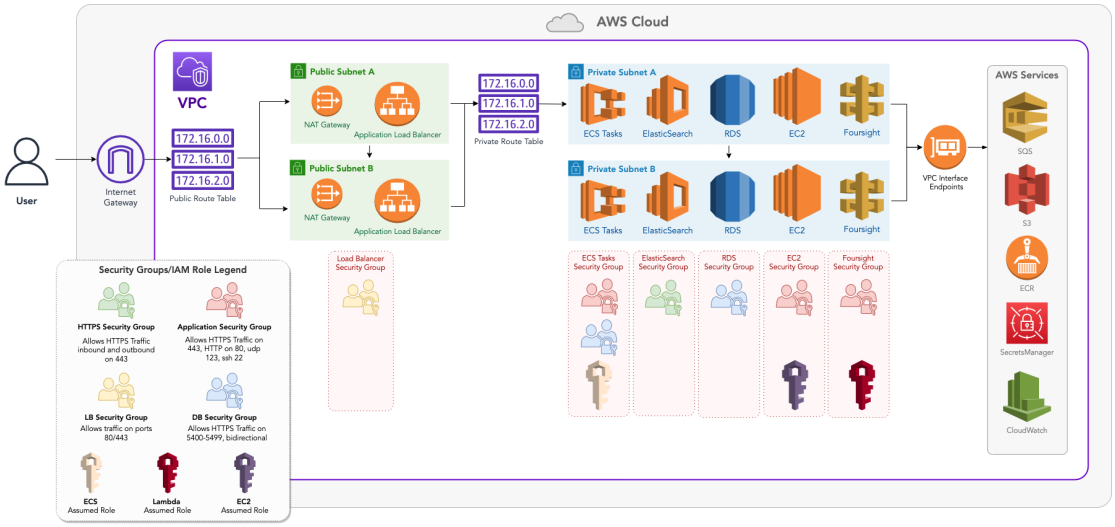

Figure 1: CGAP Infrastructure Diagram. The purpose of this diagram is to give an overview of the core functionality of the system. It is up to date as of March 2021.

CHAPTER 4

Dataflow Overview

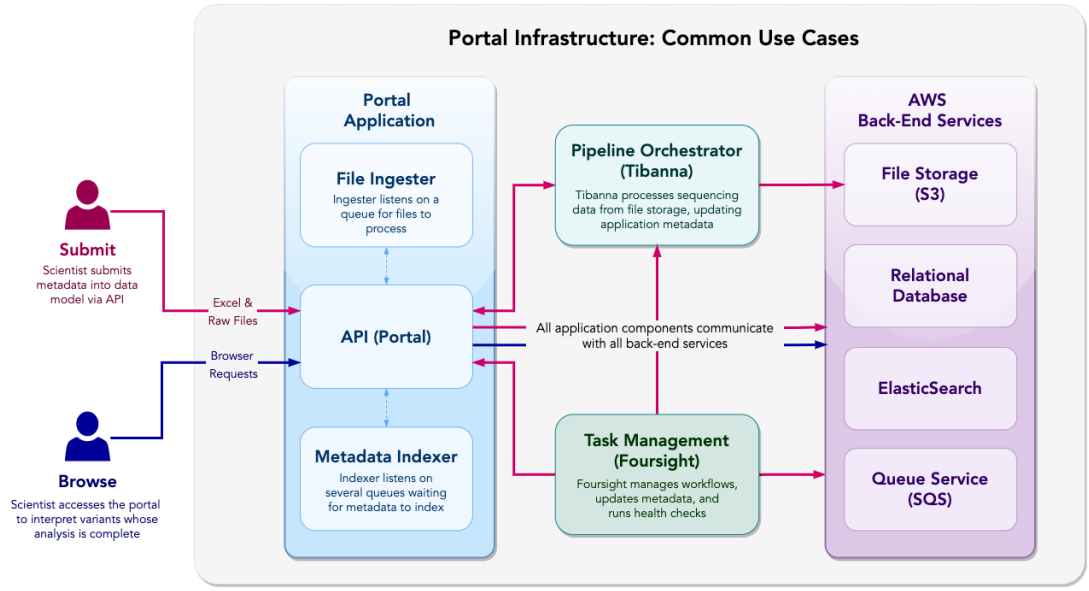

Figure 1: CGAP Dataflow/Use Case Diagram. This diagram illustrates the data browse and upload use cases, showing how that data flows through the system and which infrastructure components communicate. Note that file uploads are federated via pre-signed URL then uploaded directly to S3.

CHAPTER 5

Variant Representation

The goal is to create a useful JSON-LD Item hierarchy for working with variants. To do this, we create several different item types that are related to each other. It is possible more relationships are added, but these items form a logically related grouping so are discussed in depth in this document. Previously, these types were generated from the mapping table, but now are maintained directly.

1. GeneAnnotationField - these are annotation fields on Genes that come from Daniel's annotation DB but are processed from the gene mapping table, from this point on referred to as the gene table.
2. Gene - the gene item schema is dynamically built from the gene table. The gene items come from Daniel's annotation DB and are posted as is and validated against the schema.
3. AnnotationField - these are annotation fields on Variants that also come from Daniel's annotation DB but are processed from the variant mapping table, from this point on referred to as the variant table.
4. VariantConsequence - this is an item that encompasses information on the consequence of a variant. It has a well-defined schema and items that are generated by the wranglers.
5. Variant - the variant item schema was built from the variant table but is now directly maintained. These items represent a catalogued SNV discrepancy between an expectation set by the reference genome and an observed/measured reality that has been seen to occur in some individuals. Most commonly in our data, this is an SNV, but in principle it could be any kind of variant. To generate these items we process annotated VCF files which have passed through the bio-informatics pipeline then annotated by Daniel's annotation DB. They are again validated against the schema on post.
6. VariantSample - the variant_sample schema (and embeds) was built from the variant table but is now directly maintained. These items encompass information about the sample(s) that registered this variant and is also read from the VCF. Since multiple people could have been sequenced, the variant samples on a particular VCF do not necessary have the variant genotype (but the proband does).
7. StructuralVariant - same idea as variant, except it represents a CNV/SV call.
8. StructuralVariantSample - same idea as variant_sample, except a sample for a structural_variant.

This last step of this pipeline, which will be discussed below, is ingesting these elements. The following 5 things constitute an "ingestion version":

1. Variant Table (–> AnnotationFields)
2. Gene Table (–> GeneAnnotationFields)
3. Gene List (–> Genes)
4. (Collection of) VCF(s) (–> Variants, VariantSamples)
5. Set of VariantConsequence items

All items ingested as part of the first 4 will be given a 'MUTANNO_Version' field, which will hold the annotation DB version used to ingest. Note that only one of these 'versions' can exist in our system since the schemas are dynamically generated. The relevant items below also have the following links:

1. Variants link to Gene and VariantConsequence items
2. VariantSample links to Variant (and thus Gene and VariantConsequence)

See ingestion.py for information on code structure. Note that this structure is now deprecated in favor of direct maintaining of the schemas. The only step that runs now is VCF ingestion.

5.1 Ingestion Step 1: Gene/Variant Table Intake

The first thing we need to do is build the annotation field items and generated schemas for `variant`, `variant_sample` and `gene`. The code that does this mostly lives in `variant_table_intake.py` and is extended/repurposed in `gene_table_intake.py`. Since the tables are similar there are very few code differences needed in parsing the two tables. Note that Gene must come before Variant!

5.2 Ingestion Step 2: Gene Ingestion

Once we have our schemas, we can ingest/post the genes. These come directly from Daniel's annotation DB so not much work is required. They are posted as is, see `ingest_genes.py`.

5.3 Ingestion Step 2.5: VariantConsequences

All variant consequence items must be posted prior to ingesting the VCF, as these variants will linkTo the appropriate consequence items. These items should change very infrequently as changes to them will cause cascading invalidation and may have revision history implications.

5.4 Ingestion Step 3: VCF Ingestion

The last part of the process is ingesting the variants from VCF files and forming appropriate links. Links to Gene and VariantConsequence will happen automatically. Links from VariantSample to Variant are created manually. This requires both parsing the VCF file and formatting the items appropriately based on the schema. See `ingest_vcf.py`.

5.5 VCF Parsing Details

After producing the schemas it is time to ingest the annotated VCF. This file has a complicated structure described below. This step is written in a more object-oriented way with `VCFParser` as the main class

containing several methods specific to VCF parsing. Helper functions handle specific steps and culminate in the `run` method, which processes and entire VCF file producing all the variant and variant sample items. An overview of the steps is below.

1. Read VCF Metadata. This includes splitting VCF fields into annotation and non-annotation fields, that way we know which fields will require additional post processing.
2. Parse standard VCF fields. These are easily acquired as there is nothing special about them. The variant sample item consists entirely of these fields.
3. Parse annotation fields. These are much trickier because they are formatted differently and must be encoded a certain way to not break the VCF specification. More on this follows in the VCF specification.

5.6 Annotatated VCF Specification

Below is an outline of the annotated VCF structure with an example on how exactly it is processed.

5.6.1 VCF-Specific Restrictions

For the annotated VCF we make use of INFO fields to encapsulate our annotations. This field is part of the VCF structure and has the following restrictions on values within the field (ie: 'AC=2;VEP=1|2...' etc). 1. String format (conversion to type specified on the Mapping Table is done later) 2. No whitespace (tabs, spaces or otherwise) 3. No semicolon (delineates fields in INFO block) 4. No equals = (delineates fields in INFO block, ie: AC=2;VEP=1,2,3;) 5. Commas can only be used to separate annotation values

5.6.2 Our Restrictions

Annotation fields that should be processed as such must be marked with a MUTANNO tag in the VCF metadata as below.

Annotation fields that have MUTANNO tags must also have a corresponding INFO tag. This tag must specify the format if the annotation is multi-valued and must be pipe (|) separated. An example of each is below.

If an annotation field can have multiple entries, as is the case with VEP, these entries must be comma separated as consistent with the VCF requirements. See raw row entry below.

If an annotation field within a sub-embedded object is an array, such as vep_domains, those entries must be tilde (~) separated and no further nesting is allowed.

5.6.3 Separator Summary

1. Tab separates VCF specific fields and is thus restricted.
2. Semicolon separates different annotation fields within INFO and is thus restricted.
3. Comma separates sub-embedded objects within a single INFO field (such as VEP) and cannot be used in any other way.
4. Pipe separates multi-valued annotation fields and cannot be used in any other way
5. Tilde separates sub-embedded objects that are also arrays, such as vep_domain and cannot be used in any other way.

5.7 Parsing Example

Given these restrictions, below is a detailed walk through of how the VCF parses the annotation fields given this specification. A truncated example entry is below. Assume we are able to grab appropriate MUTANNO/INFO header information. New lines are inserted for readability but are not present in the actual file.

The first line is the VCF field header. Fields other than INFO are readily accessible. All annotation fields are collapsed into the INFO section. FORMAT and HG002 follow after INFO. The fields below are tab separated as consistent with the VCF specification. A tab separates the last part of the data above and the INFO data below.

These annotations are all single valued and are thus processed directly as strings. Conversion to actual types is done later.

Above is a VEP annotation entry that is both multi-valued and has multiple entries. To parse this we first split on the comma to get the groups. Newlines are inserted to visualize the groups. We then split on pipe since the fields are pipe separated. Even if a field is blank a pipe must be present for that field otherwise we will not be able to determine which fields go with which values. Once we have all the fields, we then go through each one and post-process. If it is an array field (not shown in this example but consistent with point 4 above) then we split again on tilde to determine the array elements, otherwise the field value is cast to the appropriate type.

CHAPTER 6

How to Provision Annotations

This section will describe how to "provision annotations", which roughly means the process of ingesting annotation related items to the portal. Note that the paths in the commands that follow may change.

6.1 Local Machine

Follow the below steps. It takes 30-45 minutes to run.

1. Startup back-end resources: `make deploy1`
2. Startup waitress: `make deploy2`
3. (If first time) Download genes: `make download-genes`
4. Load annotations: `make deploy3`

6.2 Output

The `ingestion` command uses `tqdm` to show progress bars, so you can tell what stage of the process is currently ongoing. At the end the output will look something like the below.

```
100%|| 284/284 [00:09<00:00, 30.90gene_annotation_fields/s]
100%|| 21873/21873 [20:12<00:00, 18.04genes/s]
100%|| 340/340 [00:18<00:00, 18.79variant_annotation_fields/s]
46variants [00:18, 2.44variants/s]
ERROR:encoded.commands.variant_ingestion:Encountered VCF format
error:  could not convert string to float:  '18,0,19,0'
```

The error at the end is expected with the latest VCF - if a different error occurs there should be some reasonable description. As an example, the one below looks like this:

```
ERROR:encoded.commands.variant_ingestion:Encountered VCF format
error:  could not convert string to float:  '18,0,19,0'
```

It tells you exactly which file threw the error (src/encoded/commands/variant_ingestion.py), what type of error it was (VCF format error) and what caused it (TypeError). Errors like these should be reported, along with the VCF row which threw the error (the 47th variant in the VCF since we posted 46). In this case that line has an actual VCF spec validation error.

6.3 Ingesting Additional VCFs

To ingest more VCFs with the current setup, use the `variant-ingestion` command. See `src/encoded/commands/variant_ingestion.py`.

CHAPTER 7

CGAP-Docker (Local)

With Docker, it is possible to run a local deployment of CGAP without installing any system level dependencies other than Docker. A few important notes on this setup.

- Although the build dependency layer is cached, it still takes around 4 minutes to rebuild the front-end for each image. This limitation is tolerable considering the local deployment now identically matches the execution runtime of production.
- This setup only works when users have sourced AWS Keys in the main account (to connect to the shared ES cluster).
- IMPORTANT: Take care when working with local credentials - be careful not to write them to files that end up in container images that get pushed to a public registry!

7.1 Installing Docker

Install Docker with (OSX assumed):

```
$ brew install docker
```

7.2 Configuring CGAP Docker

Use the `prepare-docker` command to configure `docker-compose.yml` and `docker-development.ini`:

```
poetry run prepare-docker -h
usage: prepare-docker [-h] [--data-set {prod,test,local,deploy}]
                      [--load-inserts] [--run-tests]
                      [--s3-encrypt-key-id S3_ENCRYPT_KEY_ID]

Prepare docker files
```

(continues on next page)

(continued from previous page)

```
optional arguments:
-h, --help            show this help message and exit
--data-set {prod,test,local,deploy}
                      the data set to use (default: local)
--load-inserts          if supplied, causes inserts to be loaded (default: not
                      loaded)
--run-tests             if supplied, causes tests to be run in container
                      (default: not tested)
--s3-encrypt-key-id S3_ENCRYPT_KEY_ID
                      an encrypt key id (default: the empty string)
```

On initial run, you will want to run with the `--load-inserts` option so data is loaded. Pass `--data-set local` to get local inserts, or `deploy` to use the production inserts.

7.3 Building CGAP Docker

There are two new Make targets that should be sufficient for normal use. To build the image locally, ensure your AWS keys are sourced and run:

```
$ make build-docker-local  # runs docker-compose build
$ make build-docker-local-clean  # runs a no-cache build, regenerating all␣
↪layers
$ make deploy-docker-local  # runs docker-compose up
$ make deploy-docker-local-daemon  # runs services in background
```

The build will take around 10 minutes the first time but will speed up dramatically after due to layer caching. In general, the rate limiting step for rebuilding is the front-end build (unless you are also updating dependencies, which will slow down the build further). Although this may seem like a drawback, the key benefit is that what you are running in Docker is essentially identical to that which is orchestrated on ECS in production. This should reduce our reliance/need for test environments.

7.4 Accessing CGAP Docker at Runtime

To access the running container:

```
$ docker ps   # will show running containers
$ docker exec -it <container_id_prefix> bash
```

7.5 Alternative Configuration with Local ElasticSearch

ElasticSearch is too compute intensive to virtualize on most machines. For this reason we use the CGAP test ES cluster for this deployment instead of spinning up an ES cluster in Docker. It is possible however to modify `docker-compose.yml` to spinup a local Elasticsearch. If your machine can handle this it is the ideal setup, but typically things are just too slow for it to be viable (YMMV).

7.6 Common Issues

Some notable issues that you may encounter include:

- The NPM build may fail/hang - this can happen when Docker does not have enough resources. Try upping the amount CPU/RAM you are allocating to Docker.
- Nginx install fails to locate GPG key - this happens when the Docker internal cache has run out of space and needs to be cleaned - see documentation on docker prune.

7.7 Docker Command Cheatsheet

Below is a small list of useful Docker commands for advanced users:

```
$ docker-compose build  # will trigger a build of the local cluster (see␣
↪make build-docker-local)
$ docker-compose build --no-cache  # will trigger a fresh build of the␣
↪entire cluster (see make build-docker-local-clean)
$ docker-compose down  # will stop cluster (can also ctrl-c)
$ docker-compose down --volumes  # will remove cluster volumes as well
$ docker-compose up  # will start cluster and log all output to console (see␣
↪make deploy-docker-local)
$ docker-compose up -d  # will start cluster in background using existing␣
↪containers (see make deploy-docker-local-daemon)
$ docker-compose up -d -V --build  # trigger a rebuild/recreation of cluster␣
↪containers
$ docker system prune  # will cleanup ALL unused Docker components - BE␣
↪CAREFUL WITH THIS
```

CHAPTER 8

CGAP-Docker (Production)

CGAP-Docker runs in production on AWS Elastic Container Service, meant to be orchestrated from the 4dn-cloud-infra repository. End users will modify the `Makefile` to suite their immediate build needs with respect to target AWS Account/ECR Repository/Tagging strategy. For more information on the specifics of the ECS setup, see 4dn-cloud-infra.

The CGAP Application has been orchestrated into the ECS Service/Task paradigm. As of writing all core application services are built into the same Docker image. Which entrypoint to run is configured by environment variable passed to the ECS Task. As such, we have 4 separate services described by the following table:

Kind	Use	Num	Spot	vCPU	Mem	Notes
Portal	Services standard API requests	1-4	Yes	4	8192	Needs autoscaling
Indexer	Hits /index at 1sec intervals indefinitely.	4 +	Yes	.25	512	Can auto-scale based on Queue Depth
Ingester	Polls SQS for ingestion tasks	1	No	1	2048	Need API to add tasks

8.1 Building an Image

NOTE: the following documentation is preserved for historical reasons in order to understand the build process.

YOU SHOULD NOT BUILD PRODUCTION IMAGES LOCALLY. ALWAYS USE CODEBUILD.

The production application configuration is in `deploy/docker/production`. A description of all the relevant files follows.

- Dockerfile - at repo top level - configurable file containing the Docker build instructions for all local and production images.
- docker-compose.yml - at repo top level - configures the local deployment, unused in production.

- assume_identity.py - script for pulling global application configuration from Secrets Manager. Note that this secret is meant to be generated by the Datastore stack in 4dn-cloud-infra and manually filled out. Note that the `$IDENTITY` option configures which secret is used by the application workers and is passed to ECS Task definitions by 4dn-cloud-infra.
- entrypoint.sh - resolves which entrypoint is used based on `$application_type`
- entrypoint_portal.sh - serves portal API requests
- entrypoint_deployment.sh - deployment entrypoint
- entrypoint_indexer.sh - indexer entrypoint
- entrypoint_ingester.sh - ingester entrypoint
- install_nginx.sh - script for pulling in nginx
- cgap_any_alpha.ini - base ini file used to build production.ini on the server given variables set in the GAC
- nginx.conf - nginx configuration

The following instructions describe how to build and push images. Note though that we assume an existing ECS setup. For instructions on how to orchestrate ECS, see 4dn-cloud-infra, but that is not the focus of this documentation.

1. Ensure the orchestrator credentials are sourced, or that your IAM user has been granted sufficient perms to push to ECR.
2. Run `make ecr-login`, which should pull ECR credentials using the currently active AWS credentials.
3. Run `make build-docker-production`.
4. Navigate to Foursight and queue the cluster update check. After around 5 minutes, the new images should be coming online. You can monitor the progress from the Target Groups console on AWS.

8.2 Tagging Strategy

As stated previously, there is a single image tag, typically `latest`, that determines the image tag that ECS will use. This tag is configurable from the 4dn-cloud-infra repository.

After a new image version has been pushed, issue a forced deployment update to the ECS cluster through Foursight. This action will spawn a new set of tasks for all services using the newer image tags. For the portal, once the new tasks are deemed healthy by ECS and the Application Load Balancer, they will be added to the Portal Target Group and immediately begin serving requests. At that time the old tasks will begin the de-registration process from the target group, after which they will be spun down. The remaining new tasks will come online more quickly since they do not need to pass load balancer health checks. Once the old tasks have been cleaned up, it is safe to trigger a deployment task through the Deployment Service.

8.3 Common Issues

In this section we detail some common errors and what to do about them. This section should be updated as more development in this setup occurs.

1. `Error: denied: User:<ARN> is not authorized to perform: ecr:InitiateLayerUpload on resource: <ECR REPO URL>`

This error can happen for several reasons:

- Invalid/incorrect IAM credentials
- IAM user has insufficient permissions
- IAM credentials are valid but from a different AWS account

Data Model

CHAPTER 9

Updating Items from Ontologies

Disorder and **Phenotype** Items correspond to ontology terms from the MONDO or HPO ontologies.

The Items are converted from owl ontology files to our json items defined by the schemas with the script `generate_items_from_owl.py`, which lives in the src/encoded/commands directory and is called from the top level directory as `bin/owl-to-items`

The script must currently be run locally. The script usage and parameters are described below.

```
bin/owl-to-items Disorder --env fourfront-cgap --load --post_report
```

Params and Options:

item_type - required Disorder or Phenotype

–env - (default = local) - environment on which to generate updates eg. fourfront-cgap the specified environment will be queries for existing items of the specified types for comparison and if *load* option is used will be the target for insert loading.

NOTE: can use *key* and *keyfile* options in place of *env* to get an auth dict from a stored set of credentials.

–input - url or path to owlfile - overrides the download_url present in script ITEM2OWL config info set at the top of the script. Useful for generating items from a specific version of an ontology - the download_url specified in the config gets the current latest version of the owl file.

–outfile - relative or absolute path and filename to write output. If you use the *load* parameter and don't specify an *outfile* you will be prompted if you wish to specify a file and as a safety backup will still generate a file with name `item_type.json` in the top level directory

–load - if param is used will use the load_data endpoint (as wrapped in the load_items function from load_items.py script) to update the database by loading the generated inserts.

–post_report - if param is used will post a Document item to the portal with name like 'item_type_Update_date-time' and the generated logfile as an attachment.

–pretty - will write output in pretty json format for easier reading

–full - will create inserts for the full file - does not filter out existing and unchanged terms - WARNING - use with care.

9.1 Processing Data Flow

- An RDF graph representation of the specified OWL ontology file is created. A specific version of an Ontology can be specified by URL or by filename (for a local owl file) - by default the URL specified in the script config gets the latest version.
- The graph is converted into a dict of term items keyed by their term_ids eg. MONDO:123456 or HP:123456 - the term is itself a dict consisting of fields whose values come from the owl. Item specific terms that come from the owl are specified in the config eg. for Phenotype the name_field is 'phenotype_name' and id_field is 'hpo_id'
- The terms/items from the file are compared to the existing Items of the specified type from the database.
  - posts are created for new Items that are not in the database
  - patches are created for existing items that have fields that have changed
  - patches to status=obsolete for existing items no longer in the file
- all the changes are logged and the json corresponding to the updates becomes part of the log
- if the *load* option is used the updates will be posted to the server using the *load_data* endpoint via the *load_items* function of `load_items.py`
- if the *post_report* option is used then the log will posted as a Document to the portal

9.2 Troubleshooting

The generation of updates and loading of inserts can be decoupled and run separately and the Document Item with the information about what happened can be generated and posted or edited manually if necessary.

Loading can be accomplished using `bin/load-items` script.

Possible most likely points of failure:

During generation of updates

- getting existing items from the database - this takes a few minutes and depends on connection to server
- downloading and processing the owl files - takes several minutes and usually depends on internet connection to external servers

During loading of updates

- typically if items fail to load there is a systematic reason that needs to be specifically resolved.
- connection issues can lead to partial loads - in this case the saved inserts should be loadable by `load_items` - the script is designed to avoid conflicts with partially loaded items.

Posting of logs

- this shouldn't fail per se but:
- if the processing fails at any point above you may have a partial log and you should have info as to where the error occurred.
- you might want to update the Document by for example, concatenating generation and load logs for a decoupled run. Or appending the successful load logs in case of interrupted loads.

Getting previous versions of ontology files

- **HPO** http://purl.obolibrary.org/obo/hp/releases/YYYY-MM-DD/hp.owl
- **MONDO** currently the versionIRI link is giving a 404 - have submitted an issue to the MONDO github.