FourFront

Sep 22, 2023

Contents


build
failing

CHAPTER 1

Overview

This is a fork from ENCODE-DCC/encoded . We are working to modularize the project and adapted to our needs for the 4D Nucleome project.

CHAPTER 2

Installation

Fourfront is known to work with Python 3.6.x and 3.7.x and will not work with Python 3.8 or greater. If part of the HMS team, it is recommended to use a high patch version, such as Python 3.7.12, since that's what we try to do with our servers, but any version of 3.7 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.

2.1 Step 0: Obtain Credentials

Obtain AWS keys. These will need to added to your environment variables or through the AWS CLI (installed later in this process).

2.2 Step 1: Verify Homebrew Itself

Verify that homebrew is working properly:

```
$ brew doctor
```

2.3 Step 2: Install Homebrewed Dependencies

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 adoptopenjdk8
$ brew install opensearch node@16
```

NOTES:

- If installation of adtopopenjdk8 fails due to an ambiguity, it should work to do this instead:

```
$ brew cask install homebrew/cask-versions/adoptopenjdk8
```

- Latest version of OpenSearch should be compatible, but if a new version is released that is incompatible the documentation may need to be updated.
- If you try to invoke elasticsearch and it is not found, you may need to link the brew-installed elasticsearch:

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

- If you need to update dependencies:

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

- If you need to upgrade brew-installed packages that don't have pinned versions, you can use the following. However, take care because there is no command to directly undo this effect:

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

2.4 Step 3: Running Make

Run make:

```
$ make build-dev  # for all dependencies
OR
$ make build      # for only application level dependencies
```

NOTES:

- 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 are running macOS Mojave (though this is fixed in Catalina), you may need to run this command as well:

    ```
    $ sudo installer -pkg /Library/Developer/CommandLineTools/Packages/macOS_SDK_
    ↪headers_for_macOS_10.14.pkg -target /
    ```

  - If you have trouble with zlib, especially in Catalina, it is probably because brew installed it in a different location. In that case, you'll want to do the following in place of the regular call to buildout:

    ```
    $ CFLAGS="-I$(brew --prefix zlib)/include" LDFLAGS="-L$(brew --prefix zlib)/
    ↪lib" bin/buildout
    ```

- If you wish to completely rebuild the application, or have updated dependencies, before you go ahead, you'll probably want to do:

```
$ make clean
```

Then goto Step 3.

2.5 Step 4: Running the Application Locally

Start the application locally.

You'll need to prepare your local python library search rules by doing the following:

```
$ python setup_eb.py develop
```

This setup only needs to be done once, even as you may do the rest of the operations that follow more than once.

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/.

CHAPTER 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
```

CHAPTER 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
```

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

CHAPTER 5

Notes on SASS/SCSS

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:

5.1 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
```

5.2 Force compiling

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

Contents

5.2.1 Overview of encoded Application

This document does not contain installation or operating instructures. See README.rst for that.

Encoded is a python/javascript application for storing, modifying, retrieving and displaying the metadata (as JSON objects) for the ENCODE project. The application was designed specifically to store metadata for high-throughput genomics experiments, but the overall architecture is suitable for any set of highly linked objects.

The "deep" backend is a simple Postgres object database. The relational database does not store any specific information about the objects but simply tracks transactions and keys. CRUD (Create/Read/Update/Delete) in this database is governed by a python Pyramid app. This python app can stand alone and provide JSON objects via GET directly from the database.

Elasticsearch is used to deeply and robustly index the entire object store and provide extremely fast read access and powerful search capability.

The Browser accessible frontend is written in ReactJS and uses the same Pyramid URL dispatch as the backend, but converts the GET request JSON into XHTML for viewing in a Web Browser.

5.2.1.1 SOURCE CODE ORGANIZATION

The top Level is organized into the following folders

- .ebextensions - contains all EB environment provisioning scripts
- bin - contains some misc scripts, such as macpoetry-build and test
- deploy - contains remaining deployment scripts
- docs - contain the source of this documentation
- examples - XXX: Unused?
- jest
- node_modules
- parts - contains WSGI process executables
- scripts - XXX: Unused?
- src - main source code

The src directory contains all the python and javascript code for front and backends

- commands - the python source for command line scripts used for synching, indexing and other utilities independent of the main Pyramid application
- docs - contains some miscellaneous docs
- locust - contains locust load testing code
- schemas - JSON schemas (JSONSchema, JSON-LD) describing allowed types and values for all metadata objects
- static - Frontend JS (components), SCSS/CSS (HTML styling), images, fonts and frontend JS libraries
- tests - Unit and integration tests
- upgrade - python instructions for upgrading old objects stored to the latest schema
- workflow_examples - XXX: document me
- workflow_test_inserts - XXX: document me

5.2.1.2 BACKEND

- Application (responds to web requests) - the main config files are *.ini in the root encoded directory.

5.2.1.3 Guts

5.2.1.4 views

The guts of the web application are in the views package. Views.views defines the Item and Collection classes that the web app will respond to via URLs like /{things}/ (returns a Collection of Things) and /{things}/{id} (retuns a Thing).

Other modules in the views package correspond to non-core views that the app will respond to.
user.py - special user objects are special access_key.py - generation/modification of access keys for programatic access search.py - constructs ES query and passes though to :9200

5.2.1.5 snovault.py

snovault.py defines the core Collection and Item classes which are the python representation of linked JSON objects and groups (collections) of linked JSON objects. It contains the business logic for updating JSON objects via PATCH and the recursive GETs necessary for embedded objects.

5.2.1.6 AuthZ

- *authentication.py*
- *authorization.py*
- *persona.py*
- **JSON data schema**

  definition Each object type has a .json schema file in /schemas. The objects are linked and embedded within each other by reference, forming a graph structure. "Mixins" are sub-schemas included in more than one object type definition. Each schema file is *versioned* and mapping an object from an older schema to a new one is called *upgrading*

  validation Objects are validated as they are POSTed or PATCHed to the application (via HTTP). Not sure when/how the validation is hooked in

  upgrading No idea

  linked and embedded objects Sorcery
- **Postgres Storage**
  - Loading
- Elasticsearch & Indexing

5.2.1.7 FRONTEND

The pyramid app handles all URL dispatch and fetches JSON objects from Elasticsearch (or optionally, the database directly). These can be either individual objects or Collections (arrays) of objects. The objects can either be "flat" with no linked objects embedded, or with some or all linked objects embedded in the response.

The scope of embedding is decided on an object-by-object bases, listed in the /src/encoded/types directory. Each object has an 'embedded' list defined, which dictates what objects will be embedded in the elasticsearch indexing process. Whole

objects can be embedded or specific fields of objects. For objects (with linkTo's in the schema) are not explicitly added to the 'embedded' list, three fields will automatically included, regardless of whether or not these are calculated properties. These are link_id, display_title, and uuid.

FOR MORE INFO ON EMBEDDING, reference docs/embedding-and-indexing.rst in snovault.

- renderers.py - code that determines whether to return HTML or JSON based on request, as well as code for starting the node subprocess renderer.js which converts the ReactJS pages into XHTML.

5.2.1.8 Use of NodeJS

5.2.1.9 About ReactJS

5.2.1.10 Component Pages

HTML pages are written in Javascript using JSX and ReactJS. These files are in src/static/components. Each object type has a component which describes how both the individual item and the collection pages are rendered. Other pages include home and search. JSX allows the JS file itself to serve like an HTML template, similar to other web frameworks.

5.2.1.11 Boilerplate and Parent Classes

- app.js
- globals.js
- mixins.js
- errors.js
- home.js
- item.js
- collection.js
- fetched.js
- edit.js
- testing.js

5.2.1.12 User Pages (Templates)

- index.js
- antibody.js
- biosample.js
- dataset.js
- experiment.js
- platform.js
- search.js

- target.js

5.2.1.13 Views and Sections (Templates)

- dbxref.js
- navbar.js
- footer.js

API

5.2.1.14 Parameters (to be supplied in POST object or via GET url parameters):

- datastore=(database|elasticsearch) default: elasticsearch
- format=json Return JSON objects instead of XHTML from browser.
- limit=((int)|all) return only some or all objects in a collection
- **Searching**
  -

5.2.2 Search Documentation

URIS

1. **http://{SERVER_NAME}/search/?searchTerm={term}** Fetches all the documents which contain the text 'term'. The result set includes wild card searches and the 'term' should be atleast 3 characters long.
   - SERVER_NAME: ENCODE server
   - term: string that can be searched accross four item_types (i.e., experiment, biosample, antibody_approval, target)

   *** TERMS ARE NOT INCLUDED until the corresponding boost values are added to the schemas of item_type *** - For example, you must add a boost of "definition" to the biosample schema for this term to be searchable for this object
2. **http://{SERVER_NAME}/search/?type={item_type}** Fetches all the documents of that particular 'item_type'
   - SERVER_NAME: ENCODE server
   - item_type: ENCODE item type (values can be: biosample, experiment, antibody_approval and target)
3. **http://{SERVER_NAME}/search/?type={item_type}&{field_name}={text}** Fetches and then filters all the documents of a particular item_type on that field
   - SERVER_NAME: ENCODE server
   - item_type: ENCODE item type (values can be: biosample, experiment, antibody_approval and target)
   - field_name: Any of the json property in the ENCODE 'item_type' schema

5.2.3 Security

In pyramids security is interwoven into the framework in a very fined grained fashion. Each view can have it's own security rules as can each object. Some basic ideas that are helpful in understanding how security works in the system are listed below.

5.2.3.1 ACL

Access Control Lists are list of three-tuples that specify security rights.The three-tuples takes the form:

These ACL's can be attached to a resource by setting the member **acl** as is done in `encoded.types.base.Item` and `encoded.types.base.Collection`. The system by Default sets up several ACL's:

Item based ACL's

- ONLY_ADMIN_VIEW
- ALLOW_EVERYONE_VIEW
- ALLOW_VIEWING_GROUP_VIEW – based on data in `award.viewing_group`
- ALLOW_LAB_SUBMITTER_EDIT – based on users Lab association
- ALLOW_CURRENT_AND_SBMITTER_EDIT – everyone can view, lab submitter can edit
- ALLOW_CURRENT – same as `ALLOW_EVERYONE_VIEW`
- DELETED – used to now show deleted objects (even though they may still be in database)

COLLECTION based ACL's

- ALLOW_SUBMITTER_ADD

5.2.3.2 Roles

There are several roles defined in the ACL's in `types\base.py`, and more can be created elsewhere in the system. Common roles are:

- group.admin – all powerful
- group.read-only-admin – can see everything
- remoteuser.INDEXER – used by Elastic Search to access all objects
- remoteuser.EMBED – used by Embed functionality to travers relationships and embed children into parent
- role-viewing_group_member – used with `ALLOW_VIEWING_GROUP_VIEW` to provide view information.
- role.lab_submitter – lab association for user to allow view / editing to appropriate data.

5.2.3.3 Permissions

Basic permissions include:

- view
- edit
- visible_for_edit – i.e. a deleted object is not visible for edit
- ['add'] – can add to a collection

5.2.3.4 Default Item permissions

By default unless specified elsewhere all `Items` get a default ACL of:

This is automatically overwritten if the `Item` has a `status` defined in `STATUS_ACL` (types-base.py(110)). For example an item with status `released` will automatically get the `ALLOW_CURRENT` ACL.

This can potentially be overwritten in a particular types.py file by overwriting the `__acl__`.

_ac_local_roles__

Just before the ACL checks an item can be assigned special roles (during traversal, i.e. during a call to the web server) based on what is defined in `_ac_local_roles__` (which by default will `add`submits_for.`and`viewing_group.<award.viewing_group>'.

5.2.3.5 User Roles

And one step earlier, i.e. before _ac_local_roles are set (which are set based on the item) a user is assigned groups based upon information stored in the user profile (see `schema\user.json`). Groups are looked up and added to the request in the `authorization.groupfinder`.

5.2.3.6 Process overview

1. Request is made to the system
2. `authorization.groupfinder` is called after user is authenticated and request is assigned the principles of user.id (or special principles for remote user / embed user / etc..). It addition prinicples are assigned for:

- `<lab>`
- `submits_for.<lab>`
- `group.submitter` – possibly
- `groups.<group>` – from user.groups
- `view_group.<group>` – from user.viewing_groups

1. Traversal happens (i.e. url is matched to view), also the custom `ac_local_roles` kicks in and adds additional principles to the request as described above.
2. Certain view functions may have an `@view_config` that adds additional security checks to see if the request can be processed (default is to use the process described in this document). These are generally declared in the types directory. i.e. `types.user` or `types.page` are good examples.

3. The request.principles are checked against the view items ACL generated from the item and its status. If the principles match and the permissions are correct the call proceeds.

5.2.3.7 Additional info

Also see `docs/auth.rst` for further information on how security works.

5.2.4 Authentication and Authorization

Background reading: Pyramid's security system.

I extend Pyramid's built in ACL based security system with my pyramid_localroles plugin so we can map permissions to roles (e.g. 'role.lab_submitter') rather than directly to users.

For more on roles and local roles see:

- http://docs.zope.org/zope2/zope2book/Security.html#different-levels-of-access-with-roles
- http://www.sixfeetup.com/blog/basic-roles-and-permissions-in-plone
- https://www.packtpub.com/books/content/plone-4-development-understanding-zope-security

5.2.4.1 Authentication

An authentication policy identifies who you are, returning a user id. We use pyramid_multiauth to extract authentication from any of Persona, session cookies, or HTTP basic auth (access keys).

5.2.4.2 Authorization

From the authenticated user id, the groupfinder in authorization.py maps the user id to a number of "principals", user or group identifiers. We lookup the user object and add groups based on the properties:

- groups [<string>..] - global groups like 'admin'. Generates: *group.admin*.
- submits_for: [lab..] - allow editing based on object.lab property. Generates: *submits_for.<lab-uuid>*.
- viewing_groups: [<string..>] - allow viewing of in progress data based on object.award.viewing_group (ENCODE, GGR, REMC.) Generates: *viewing_group.ENCODE*.

Views are protected by permissions (*view*, *edit*, etc.)

When you PUT to /experiment/ENCSR123ABC/ then Pyramid will traverse to the experiment object (see: Location aware resources) and lookup a view for to PUT which is protected with the *edit* permission.

At this point my pyramid_localroles plugin steps in and extends the authenticated principals passed to the ACLAuthorizationPolicy (the global groups that apply across the whole site) with location aware local roles such as *role.lab_submitter* and *role.viewing_group_member* by reference to the *__ac_local_roles__* method (base.py) of the context object which returns a mapping based on the context object's 'lab' and award property, e.g:

```
{
  'submits_for.<context-lab-uuid>': ['role.lab_submitter'],
  'viewing_group.<context-award-viewing_group>'': ['role.viewing_group_member
↪'],
}
```

The ACL authorization policy will then lookup the Access Control List on the experiment object (the 'context') by looking at its *__acl__* property/method, and then the *__acl__* property/methods of its parents (the /experiments collection and the root object.) We define an *__acl__* method on the EncodedRoot object (root.py), Collection and Item objects (base.py.) The *__acl__* method for an Item returns a different ACL list depending on the object's 'status'. This way we allow lab submitters to edit their own 'in progress' objects but not 'released' objects.

5.2.4.3 Permissions

- add
- add_unvalidated (admin)
- edit
- edit_unvalidated (admin)
- expand (system)
- forms - who can see forms
- impersonate (admin)
- import_items (admin)
- index (system)
- list
- search
- submit_for_any (admin)
- view
- view_details - protection of user contact information
- view_raw (admin)
- visible_for_edit - hiding deleted child objects from edit

Permissions of items are tied to the statuses of items. We have 8 statuses for most items (there are exceptions like file and publication)

- Current : Everyone can view, admin can edit
- Released : Everyone can view, admin can edit
- Revoked : Everyone can view, admin can edit
- Deleted : Nobody can view, admin can edit
- Replaced : Everyone can view, admin can edit
- Obsolete : Nobody can view, admin can edit
- In review by lab : Lab members can view, submitter can edit
- submission in progress : Project members can view, submitter can edit
- Released to project : Project members can view

This gnu grep expression will extract a list of permissions (brew tap homebrew/dupes; brew install grep):

```
$ ggrep --no-filename -roP "(?<=permission[=(][']\"])[^'\"]+" src/ | sort |␣
↪uniq
```

5.2.5 FF-Docker (Local)

With Docker, it is possible to run a local deployment of FF 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: Do not upload the local deployment container image to any registry.

5.2.5.1 Installing Docker

Install Docker with (OSX assumed):

```
$ brew install docker
```

5.2.5.2 Configuring FF 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

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), not␣
↪typically used for FF
```

Note that you must additionally set `GLOBAL_ENV_BUCKET=foursight-prod-envs`, which will be passed to the container to resolve environment information. 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.

5.2.5.3 Building FF 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.

5.2.5.4 Accessing FF Docker at Runtime

To access the running container:

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

5.2.5.5 Alternative Configuration with Local ElasticSearch

ElasticSearch is too compute intensive to virtualize on most machines. For this reason we use the FF 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).

5.2.5.6 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. This can be done easily from Docker Desktop, opening the Settings and then accessing the Resources panel. Try 6 CPUs and >8 GB RAM.
- 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.

5.2.5.7 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)
```

(continues on next page)

(continued from previous page)

```
$ 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
```

5.2.6 FF-Docker (Production)

FF-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. Note that builds should be automated through AWS CodeBuild. For more information on the specifics of the ECS setup, see 4dn-cloud-infra.

The FF 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

5.2.6.1 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
- install_nginx.sh - script for pulling in nginx
- fourfront_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.

5.2.6.2 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.

5.2.6.3 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

5.2.7 Database Documentation

The (encodeD) system uses a Postgres implementation of a document store of a *JSON-LD* object hierarchy. Multiple views of each document are indexed in *Elasticsearch* for speed and efficient faceting and filtering. The JSON-LD object tree can be exported from Elasticsearch with a query, converted to *RDF* and loaded into a *SPARQL* store for arbitrary queries.

5.2.7.1 PostgreSQL RDB

When an object is POSTed to a collection, and has passed schema validation, it is inserted into the Postgres object store, defined in *storage.py*.

There are 7 tables in the RDB. Of these, *Resource* represents a single URI. Most Resources (otherwise known as Items or simply "objects" are represented by a single *PropSheet*, but the facility exists for multiple PropSheets per Resource (this is used for attachments and files, in which the actual data is stored as BLOBS instead of JSON).

The *Key* and *Link* tables are indexes used for performance optimziation. Keys are to find specific unique aliases of Resources (so that all objects have identifiers other than the UUID primary key), while Links are used to track all the JSON-LD relationships between objects (Resources). Specifically, the Link table is accessed when an Item is updated, to trigger reindexing of all Items that imbed the updated Item.

The *CurrentPropSheet* and *TransactionRecord* tables are used to track all changes made to objects via transactions.

Local Machine Development

5.2.7.2 Booting Up Local Database

The `bin/dev-servers` command, required as part of the boot-up process (see the repo README) completely drops and restarts a local copy of the PostegreSQL server instance and database. Script posts all the objects in tests/data/inserts (plus /tests/data/documents as attachments). Then indexes them all in local elastic search. The server instance and 'postgres' database are both destroyed when you kill the dev-servers process.

This temporary PostgreSQL database exists in the filesystem in your Unix-based system's `/tmp/snovault/pgdata` folder - which may be connected to as the hostname. The database created is named 'postgres', with an admin user also named 'postgres', and also should be accessible via localhost port 5432. No password is required.

By default, insert test data defined in Fourfront is loaded into the local database. See the inserts documentation for more information.

Backup & Loading of Production Database

5.2.7.3 Purpose

There may be many reasons to back-up live database data. At minimal, we should have periodic back-ups in case the production environment and database melt due to Murphy's Law. Another reason may be to load live production data to local environment for more thorough testing when the test inserts might not be complete enough.

5.2.7.4 Prerequisites

Software

PgAdmin is recommended for performing back-ups, as well as other PgSQL-centric tasks. PgAdmin provides a GUI for interacting with your database(s), and also allows you to explore the PropSheets and other database data. You also need to make sure you have Amazon's **Elastic Beanstalk Command Line Interface** (**EB CLI**), installed and configured with your Amazon key, as well as have a copy of the *4dn-encode* private key in your */Users/YourName/.ssh* folder. We'll need to create a SSH tunnel through an Amazon EB/EC2 instance to our live production database - which is not accessible from the public internet for security reasons.

Configuration

Make sure your EB CLI is working and you've been able to SSH into an EC2 instance. You likely do this by running something like:

```
eb ssh -n 1 --custom "ssh -i /Users/alex/.ssh/4dn-encode.pem"
```

where `4nd-encode.pem` is your copy of the 4dn-encode private key. You'll also need the hostname, port, username, database, and password (or connection string) of the RDS (Amazon term for database instance/server) where the live database is located, which may be obtained by logging into the AWS console and looking at the environment variables configured for the Elastic Beanstalk environment whose database you want to access. The hostname will likely resemble `fourfront-webprod.co3gwj7b7tpq.us-east-1.rds.amazonaws.com:5432`. We aren't going to write what the username, database, and password will resemble in a public document.

5.2.7.5 Back It Up

Once you have your prerequisites, do the following:

1. In a dedicated Terminal window, create an SSH tunnel via *eb ssh* command to the public RDS database. The command will look like this:

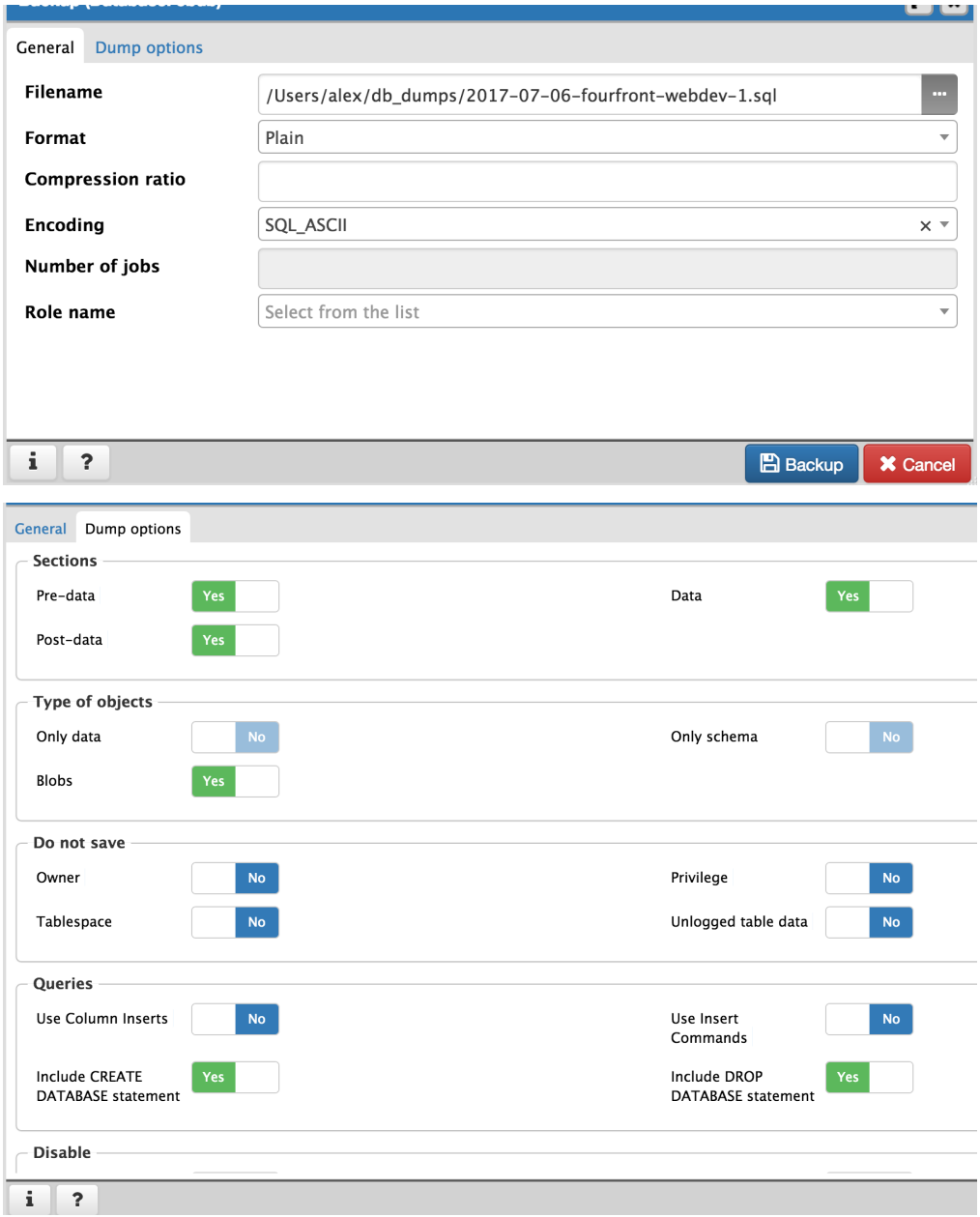
   ```
   eb ssh -n 1 --custom "ssh -i /Users/alex/.ssh/4dn-encode.pem -L␣
   ↪5999:fourfront-webprod.co3gwj7b7tpq.us-east-1.rds.amazonaws.com:5432"
   ```

   By using the '-L' argument, you create a tunnel from your local port 5999 to a remote host:port on the EC2 instance you're connecting to. Replace path to your 4dn-encode private key and host:port of remote RDS accordingly. *N.B.* We need to use `eb ssh` rather than plain `ssh` because `eb ssh` tells Amazon to temporarily open port 22 (for SSHing) which would otherwise remain closed for security reasons.

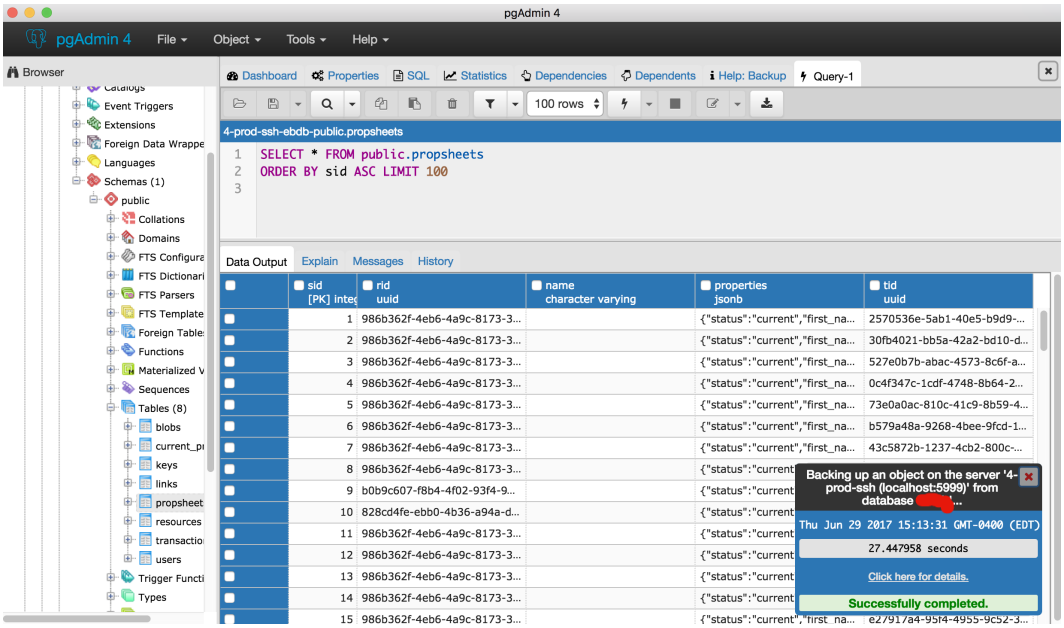
   1. Open *PgAdmin* and, if not yet done, create a new 'Server' connection and call it "SSH to fourfront-webprod db" or something relevant. Make sure the hostname is *localhost:5999*, as we'll be utilizing the SSH tunnel we created in step 1. Use the database, username, and password that are defined in the AWS EB environment variables configuration.
   2. On the left tree-view pane of PgAdmin, should see the live RDS server, hopefully connected. Expand the server until see the database which have connected to. Right click on the database, and select "Backup…" from the drop-down menu.
   3. It is important to set the right backup options. Your filename isn't important but should make sense. The suggested format is `YYYY-MM-DD-ENVNAME-I.sql`, e.g. *'2017-07-19-fourfront-webprod-2.sql'*'. The following options are important (spread across both tabs)

      - Select "Plain Text" for format.
      - For encoding, select SQL_ASCII or similar. Your luck with UTF-8 may vary.
      - Under "Dump Options" tab, ensure the following are set to "Yes":
        - Pre-Data
        - Post-Data
        - Data
        - "Include DROP DATABASE statement"
        - "Include CREATE DATABASE statement"

- "With OIDs"
- (Optional) "Use INSERT Commands"
- Other options may be left on default or adjusted to your needs.



2. Click "Backup". PgAdmin should pop up a little box on bottom right of their GUI showing time elapsed and then a success or error message. This should take about 30 seconds (or longer) as of 2017-07-06.



3. Navigate to your newly backed up SQL file. There it is! Remember to disconnect the server and SSH tunnel when done.

5.2.7.6 Load It In

No point in backing up data if can't get it to work again. Even if backing up for the sole sake of having back-ups, an untested back-up is no back-up at all.

If you backed up your .SQL file with no issues, you should be able to easily import the data back into production without issue by SSH tunneling to the production RDS again and running the .SQL file against the production database with the *psql* command. Don't try this without reason, though, for the sake of production data stability.

If want to import into your local, there are a few extra steps needed, and a few things to keep in mind to keep your machine performant.

1. With your local environment shut down, run:

```
bin/dev-servers development.ini --app-name app --clear --init --load
```

   as usual, but do not run `bin/pserve` yet. This will boot up your local PostgreSQL server and database but not launch the web app yet.

2. In your favorite text editor ***which can handle large files***, open the SQL file which you backed up earlier. Do a search & replace for the user (from EB environment variable) and replace all instances with 'postgres', to match the user used to connect to your local server. You can also search & replace all instances of the database name -**if**- you want to change it from production database name (not suggested). Assuming your database name in SQL file is different than 'postgres' (database name of database created by local environment), you will be creating another database on the same local PostgreSQL server, alongside the database with your test inserts (initially loaded in *bin/dev-servers* and named 'postgres' (not to be confused with user name of same value)).

3. Run:

```
psql -h /tmp/snovault/pgdata -U postgres -w postgres -a -f "/Users/alex/
↪db_dumps/2017-06-29-fourfront-webdev-1.sql"
```

to run SQL file against your PostgreSQL server instance, replacing the SQL file path and name with your own. This will create and populate another database with your backed up data, alongside the one created and populated with test inserts by *bin/dev-servers* command.

4. Open up your *development.ini* file. Create a copy of it you'd like, or just adjust locally and don't commit. Make the following changes:

   - Comment out the existing `sqlalchemy.url` option, and replace it with:

     ```
     sqlalchemy.url = postgresql://postgres@:5432/DATABASE_NAME?host=/tmp/
     ↪snovault/pgdata
     ```

     where DATABASE_NAME is database name of the database you loaded in with your SQL backup file. This will ensure you connect to your backed-up database when you boot *bin/pserve* instead of the test inserts database from *bin/dev-servers*.

   - Under both `[composite:indexer]` & `[composite:file_indexer]` sections, add the following:

     ```
     timeout = 64800
     ```

     By default, the indexer runs once a minute, and on local machine, it runs for 45 minutes. While running, the indexer uses a lot of energy and is very likely to overheat laptops – especially if running continuously. It may drain your battery faster than you can charge it. Adjusting the auto-indexing timeout to 48 hours instead of one minute alleviates most of this pain except for initial indexing-upon-bootup.

   - Save (or save copy of) adjusted development.ini file.

5. Finally, run `bin/pserve development.ini` (if created a copy of development.ini, replace "development.ini" in command with your .ini filename). It should start indexing through tens of thousands of entries. Grab lunch while your laptop fans learn how to fly. Return to a local portal running with production data. Remember to revert your development.ini when want to load in test inserts instead of production data.

Afterthoughts

In lieu of PgAdmin, may use the command-line `pg_dump` tool to connect to production database (over SSH tunnel) and save output to SQL file. Ensure the same configuration (ASCII, no compression, CREATE/DROP DATABASE command, . . .) is set as for PgAdmin when running it.

Eventually, creating a shell or Python script to automate backup (and potentially import) may become a task, wherein the backup script could then perhaps be run on a scheduled basis.

5.2.8 Higlass Visualization

This document explains the end to end behavior of the visualization endpoint.

5.2.8.1 API Call

Make a POST request to `add_files_to_higlass_viewconf/`. The fourfront server will return the viewconf used to create Higlass Items.

Payload

higlass_viewconfig

A base viewconf to add the files to. If not provided or null, the server uses a default "blank" viewconf.

files

A list of file accessions. Each file will be added to the viewconf.

height

Expected height for all of the tracks. If not provided, the default height is 600 pixels.

firstViewLocationAndZoom

An array of 3 numbers. These correspond to the coordinates of the first view, as well as its zoom level.

The first 2 numbers indicate the center of the highlighted data, while the final number notes how zoomed in the view is.

The zoom level relies on d3 library's implementation, so if you want to experiment with it, find some already existing viewconfs and edit the location locks.

If not provided, the view will point at the center of the domain, with the zoom level covering the entire domain range.

remove_unneeded_tracks

If there are no 2D Higlass tracks (for example, no mcool files,) and this to true, all of the left side tracks will be removed. By default, this is false.

Example payload

```
{
"files": ["4DNFIWG6CQQA", "4DNFIZJB62D1", "4DNFIWQJFZHS", "4DNFI9UM7MDC",
↪"4DNFIZMTKWDI", "4DNFIC624FKJ"]
}
```

Creates a new viewconf. All of the files are checked to make sure they have the same genome assembly. Here's a sample output.

```
{
  "success": true,
  "errors": "",
  "new_viewconfig": {
    <truncated>
  },
  "new_genome_assembly": "GRCh38"
}
```

</details>

You still need to POST or PATCH the `new_viewconf` object to `higlass-view-configs/` to create/edit a Higlass item.

Viewconf limits

1D tracks only

- Gene annotation files are always first, chromsizes files are always last
- Otherwise, each track is added to the top in order they are listed.
- There are no left side tracks (unless the view had a 2D track before. Use `remove_unneeded_tracks` in that case.)

Single 2D track

- The 1D tracks on top will be mirrored on the left side.
- Only 1 2D track in a given view. It will be in the center of the viewconf.
- A chromsizes grid is added on top of the 2D track.

Multiple 2D tracks

- Only 1 2D track per view.
- Adding another will copy the first view, replacing the track.
- All views are "locked" so scrolling or zooming one view will scroll/zoom the others.
- No more than 6 views per viewconf. If there are more than 2, the view will create a second row to add the third view.

Errors and Issues

All files must have a uuid, higlass_uid and genome assembly

The POST still returns a 200 status, but the `errors` field will be non-empty and `success` will be false.

Make sure all of files have the same genome assembly

If the files have mismatched genome assemblies, you'll get an error.

```
{
  "success": false,
  "errors": "Files have multiple genome assemblies: GRCh38: 4DNFIWG6CQQA,␣
↪4DNFIZJB62D1; GRCm38: 4DNFIU37KWB1, 4DNFIU37KWB1, 4DNFIU37KWB1,␣
↪4DNFIU37KWB1, 4DNFIU37KWB1, 4DNFIU37KWB1",
  "new_viewconfig": null,
  "new_genome_assembly": null
}
```

Fourfront display adjustment

By default, Higlass Items are 600 pixels high. But Experiment Set pages allow 300 pixels for Higlass Items. Front end javascript will dynamically resize a copy of the viewconfig to fit.

- 2D tracks adjust their height automatically, so they are not modified.
- If there are 1D and 2D tracks in the viewconf, the 2D track is set to 2/3 of the container height.
- If there are more than 2 views, the container halves the relative amount of height to work with.
- 1D tracks will be scaled so they maintain the relative amount of space in the new container.

Foursight Higlass checks

Foursight uses the Fourfront endpoint to create and update HiglassItems. All of the checks work on a file or experiment set.

5.2.8.2 Foursight finds reference files

Foursight reads the genome assembly from the source files, and gets the relevant chromsizes and beddb files.

5.2.8.3 File Higlass Items

Foursight looks for files with Higlass uids and genome assemblies. There are additional queries used to further filter, based on the Foursight check.

With the file and the reference files Foursight calls the Fourfront API, gets the `new_viewconf` and creates a new Higlass Item. The File's static_content section is updated so it refers to the uuid of the Higlass item.

5.2.8.4 Experiment Set (Processed Files) Higlass Items

Foursight looks for ExpSets with:

- A `processed_files` section with files with Higlass uids and genome assemblies.
- At least one `experiments_in_set` object with a `processed_files` section with files with Higlass uids and genome assemblies.

And then applies queries to filter further, based on the Foursight check.

All of the files in the processed_files section with Higlass uids and genome assemblies are combined with the reference files to make or update a Higlass Item. The ExpSet's static_content is updated so the `tab:processed-files` section uses the new Higlass Item.

5.2.8.5 Experiment Set (Other Processed Files aka Supplementary Files) Higlass Items

The opf section is a bit more complicated because each group has its own Higlass Item.

Foursight looks for ExpSets with a `other_processed_files` section. For each group it sees which groups are worth updating:

- There are files with Higlass uids and a genome assembly

- There is no Higlass Item for this group
- OR The files have been updated after the Higlass Item (the Higlass Item is at least `minutes_leeway` minutes older)

Each opf group in the ExpSet (not the `experiments_in_set.other_processed_files` section) is updated.

```
{
 "files" : [ "<list of file accessions, OR an empty array, see below>" ],
 "title" : "<Name of the opf group>"
 "higlass_view_config" : "<higlass item uuid>"
}
```

If the files come from `experiments_in_set.other_processed_files`, the `files` array is empty. Otherwise it contains all of the `experiment_set.other_processed_files` used.

5.2.9 Loading Inserts

Fourfront has a set of json insert files that are used to load data in various environments. These are loaded using `bin/load-data`, which calls the functions defined in src/encoded/loadxl.py.

The behavior of load-data depends on the current Fourfront environment and the `snovault.load_test_data` setting in the used .ini file. This documentation goes into some detail on those options; to read about which inserts are used, see this documentation.

5.2.9.1 bin/load-data

Main command for loading insert data. Example usage is:

The arguments are as follows:

- **config_uri**: required. Path to the .ini configuration file
- **–app-name**: Pyramid app name in configfile, usually “app”
- **–access-key**: if “s3” (default), will create and upload a new admin access key to s3. Otherwise, if “local”, will build a local `keypairs.json` file and add the key to that
- **–drop-db-on-mt**: if True and the Fourfront environment is “fourfront-mastertest”, will drop the DB before loading inserts for a fresh test DB
- **–prod**: boolean flag that must be used to run load inserts on either “fourfront-webprod” or “fourfront-webprod2” environments

5.2.9.2 App configuration

The load function used is defined under `snovault.load_test_data` in the .ini configuration file. For local usage, this is `development.ini` and the default load function used is `encoded.loadxl:load_local_data`. For production environments, the value of this setting should be set as the `LOAD_FUNCTION` environment variable. This will probably be either `load_prod_data` for staging/data environments or `load_test_data` for test environments. Again, these configuration values correspond to the functions used in loadxl.py.

5.2.10 Dependencies and Invalidation

Keeping elasticsearch in sync.

The /_indexer wsgi app (es_index_listener.py) drives the incremental indexing process. When a new transaction is notified by postgres (or after 60 seconds) it calls the /index view (indexer.py) which works out what needs to be reindexed. The actual reindexing happens in parallel in multiprocessing subprocesses (mpindexer.py.)

When rendering a response, we record the set of embedded_uuids and linked_uuids used.

- `embedded_uuids` are those objects embedded into the response or whose properties have been consulted in rendering of the response. Any change to one of these objects should cause an invalidation. (See `Item.__json__`.)
- `linked_uuids` are the objects linked to in the response. Only changes to their url need trigger an invalidation. (See `Item.__resource_url__`.)

When modifying objects, event subscribers keep track of which objects where updated and their resource paths before and after the modification. This is used to record the set of `updated_uuids` and `renamed_uuids` in the transaction log. (See indexing.py.)

The indexer process listens for notifications of new transactions. With the union of updated_uuids and union of renamed_uuids across each transaction in the log since its last indexing run, it performs a search for all objects where embedded_uuids intersect with the updated_uuids or linked_uuids intersect with the renamed_uuids. The result is the set of invalidated objects which must be reindexed in addition to those that were modified (recorded in updated_uuids.)

Where an object's url depends on other objects – `Page` whose url includes its ancestors in its path, or `Target` whose url includes a property from its referenced organism – we must ensure that linked_uuid dependencies to those other objects are recorded in addition the object itself when linked. (See `Page.__resource_url__` and `Target.__resource_url__`.)

5.2.10.1 Total Reindexing

Cases can arise where a total reindexing needs to be triggered. >curl -XDELETE 'localhost:9200/encoded/meta/indexing' will specifically force it.

localhost:9200/encoded/meta/indexing stores the document that keeps track of incremental indexing. The indexer script checks for that document when deciding between full index and indexing only the recently invalidated documents. It has the benefit of keeping the old-yet-to-be-indexed data online, especially if it's a production instance.

Alternatively, >curl -XDELETE 'http://localhost:9200/encoded/' will delete the entire index along with the mapping information for schema objects. Although it does trigger indexing, missing mapping information makes the documets unsearcheable. Mapping in elasticsearch describes how each field of each object should be tokenized/analyzed/indexed for searching.

5.2.10.2 Back references (rev-links)

In a parent-child relationship, it is the child object that references the parent object. A parent response often renders a list of child objects, and that list my be filtered to remove deleted or unpublished child objects.

We want to ensure that parent responses are invalidated when a child object's state changes, so that it would now be included in its parent's list of child objects when it was not before. A parent response must therefore include all *potentially* included child objects in its `embedded_uuids`, which is done by accessing the child status property through the `Item.__json__` method.

We must also invalidate a parent response when a new child is added (either a new object of changing the parent referenced.) This is done adding the parent uuid to the list of updated_uuids recorded on the transaction adding/modifying the child. (See indexing.py `invalidate_new_back_revs`.)

SEE ALSO rev-links.md for more information about how to setup rev links

5.2.10.3 Isolation level considerations

Postgres defaults to its lowest isolation level, READ COMMITTED: http://www.postgresql.org/docs/9.3/static/transaction-iso.html

For invalidation of back references of new child objects only READ COMMITTED isolation is necessary as invalidated back references are calculated from the updated objects properties.

However, writes must be at least REPEATABLE READ in order for overalapping PATCHes to apply safely.

During recovery indexing uses READ COMMITTED isolation. Indexed objects may be internally inconsistent if there are concurrent updates to embedded objects. But indexing is still eventually consistent as any concurrent update will invalidate the object and it will be reindexed later.

To avoid internal possible internal inconsistancies of indexed objects, SERIALIZABLE isolation is required. It is used once it becomes available when recovery is complete.

5.2.11 Local Deployment Troubleshooting

NOTE

If you had problems with your local deployment, and found solutions to them, please document them here. Please include software versions, and date

---

5.2.11.1 20190218 Pillow 3.1.1 install error on Mac 10.14.3, Xcode 10.1 (command line tools 10.1 10B61) - Koray

error message

```
--enable-zlib requested but zlib not found, aborting
```

I switched to Mojave, decided to do a fresh install of ff, and updated Xcode from appstore, run `xcode-select --install` to update (you might need to restart computer after installing xcode, run xcode, and agree to the terms). It turns out the new (Mojave) Xcode Command Line tools no longer installs needed headers in /include. This did the trick for me

```
sudo installer -pkg /Library/Developer/CommandLineTools/Packages/
macOS_SDK_headers_for_macOS_10.14.pkg -target /
```

for more info https://github.com/pyenv/pyenv/issues/1219

5.2.11.2 20190219 Server does not start on Mac 10.14.3, Xcode 10.1 (command line tools 10.1 10B61) - Koray

error message

I and Carl tried various things (rebuilds, re-linking brew ...) but it did not help. At the end I did the following, I guess deleting the folder was the key.

- delete all brew elastic search versions
- delete the folder /usr/local/etc/elasticsearch/
- reinstall elasticsearch
- rebuild

#

5.2.12 Static Pages

Most static pages content - unless hard-coded for the front-end (in case of custom interactivity, etc.) - exists in HTML or Markdown files in the repository, in an S3 bucket, or in-line within an insert. Contents of a page is an array of linkTo StaticSection items loaded in the same way as other Items, and exists in the "content" property on the Page Item. The "name" property of the Page Item becomes the static page's path where it may be viewed.

In the `/src/encoded/tests/data/[..]/page.json` file, an insert defining the page available at "help/submitting/getting-started" might be in this form:

```
[...
{
    "title" : "Getting Started with Submissions",
    "name" : "help/submitter-guide/getting-started",
    "content" : [
        "help.submitter-guide.getting-started.introduction",
        "help.submitter-guide.getting-started.metadata-structure"
    ],
    "table-of-contents" : {
        "enabled" : true,
        "header-depth"  : 4
    }

}
...]
```

Notice the "content" property, which links to StaticSection Items which might look like the following in `/src/encoded/tests/data/[..]/static_section.json` inserts:

```
[...
{
    "name" : "help.submitter-guide.getting-started.introduction",
    "body" : "To get started, ...",
    "toc-title" : "Introduction",
    "options" : {
        "filetype" : "md"
    }
}, {
```

(continues on next page)

(continued from previous page)

```
    "name" : "help.submitter-guide.getting-started.metadata-structure",
    "file" : "/docs/public/metadata-submission/metadatastructure1.html",
    "title" : "Metadata Structure"
},
...]
```

What the above configuration objects says, is for the back-end to enable a page route "help/submitter-guide/getting-started", and at that route to return some JSON which has two sections - one with no title visible on page but with one in the table of contents ("Introduction"); and one with the same title visible for both table of contents and on page ("Metadata Structure"). If do not include "title" nor "toc-title", or have "title" set to null without a "toc-title", the section (& any children) will be excluded from the table of contents (but not the page). A Page title is mandatory (but not StaticSection title).

Importantly – if *two or more* StaticSections have titles or toc-titles defined on a page, then **ALL** section titles for that page will be visible in the table of contents, even if they do not exist, as otherwise header depths within different sections cannot properly/automatically align. If e.g. have a page with 3 sections, and two of them have titles, then the third section (without a title) will get an auto-generated title based off of its name (dashes replaced with spaces and capitalized) to be shown in the Table of Contents.

The section content will be the raw contents of the file located at `file` property (which maybe a remote location). The entirety of the "table-of-contents" object is sent across to the front-end to be used as configuration options for the table of contents. If "enabled" is set to false in this configuration, the page rendered on front-end will have just a single wider pane with all the content in lieu of a Table of Contents.

Section content is parsed based on the optional `options.filetype` field, which defaults to plain HTML. If a file is used as source of content (whether in repo or S3 bucket), this `options.filetype` is unnecessary as it is obtained from the file ending.

5.2.12.1 HTML Content

Is excluded from Table of Contents except for Section title (if any).

For HTML content (filename with .html extension or `object.filetype` not filled or set to 'HTML'), no further parsing is performed for table of contents (aside from showing the table of contents if `PageItem["table-of-contents"]["enabled"] == true`). HTML content is simply inserted into sections of the page (under its section title, if any set), along with corresponding entries for the sections in Table of Contents. First-level ToC links navigate you in-page to top of section. Headers within the HTML content do not currently get parsed and added to Table of Contents (though this can be implemented at some point).

5.2.12.2 Markdown Content

For any Markdown content (filename with .md extension object.filetype set to 'md'), for each section of content (contents of file from 'filename'), the Table of Contents front-end script "looks" through the parsed Markdown content to gather next-decrement-level headers up until a header of same level as current ToC entry, and then dynamically generates links for those next-level headers in the Table of Conents which would navigate you in-page to that Markdown header.

This functionality may be controlled by the `header-depth` field in "table-of-contents" configuration. Only children headers as low as `header-depth` will be included in the ToC so that small headers may be excluded. By default, this is 6, as headers in HTML markup go as 'deep' as 6 (h1, h2, h3, h4, h5, h6). To only show section titles and no Markdown headers within the Table of Contents for a page, it should be enough to set `header-depth` to 0 or 1.

5.2.12.3 Text/String Content

For a section, can also define `file` to refer to a .txt file or have a plain-text `body` field (`object.filetype` == "txt"). It will be treated more or less like plain HTML but be slightly better implemented and safer for use on front-end.

Interactive React Component Placeholders (for front-end developers)

Sometimes, you may want to put some dynamic element onto a static page, but don't want entire static page to be defined on the front-end. The `/help` page is a perfect example, as the vast majority of the content is in Markdown files, but there is an interactive slideshow that exists halfway down the page. For this, we create a "Text/String Content" section ("content" property instead of "filename" property), and in the content, put in a "placeholder" string. In such cases you will almost always want to exclude "title" property or set it to null, so the interactive element doesn't appear in Table of Contents.

The placeholder string should look like this (displayed in context of section definition):

```
... {
    "filename" : "carousel-place-holder",
    "content" : "placeholder: <SlideCarousel />"
}, ...
```

It will be the word "placeholder", followed by a colon, followed by any string you want – though React JSX syntax is reccommended for clarity. On the front-end, in the view or template React component which handles that particular static page route, there must exist a function named *`replacePlaceHolder(placeholderString)`*. This function will accept the string after `placeholder:`, with spaces removed, and should return a valid JSX element. For clarity, it is suggested to have the placeholder string be the same as the React/JSX component output of that function for that string. Having replacePlaceHolder() allows us to avoid security risks inherent in calling 'eval(...)'.

Best Practices

- DO split Pages into multiple StaticSections with proper title for each, if possible, rather than having Page that has just one big long Markdown section/file.
  - This will allow each section to be re-used in other places & apply permissions to each section.
  - If there is only one or less sections with a title (e.g. could have multiple sections all with no titles or just one big long section), then the ## (h2) headers get promoted as if they were Section headers in TableOfContents. However, styling within the page itself will remain as Markdown h2 header (not section header). H1 (#) headers are reserved for Page titles and are not currently supported within (our parsing of) Markdown.
  - If have *2*+ static sections with titles, all sections and their titles — even if nonexistent — will be displayed in TableOfContents. If there's a section for which title doesn't exist, title will default to (JS version of) `" ".join([ word.capitalize() for word in section.link.split("-") ])` where `section.link` is last bit of StaticSection name (e.g. "path.to.section.lorem-ipsum-1" => "Lorem Ipsum 1").
- If are going to edit Pages/Sections through Fourfront UI (rather than using a Markdown/text editor & then adding to inserts) — then is a good idea to keep inserts up-to-date in order to make local development + testing simpler as well as provide an extra source of backups.
  - Our primary mission isn't to maintain/support a custom content management system so having a concrete outside-of-db representation of static pages I think is desireable.

  - There is now a command called `bin/export-data` which can be used to export Page and StaticSection inserts into JSON files. Examples: .. code-block:: bash

    bin/export-data "https://data.4dnucleome.org/search/?type=Page&limit=all" -u ACCESS_KEY_ID -p ACCESS_KEY_SECRET > new_page_inserts_file.json bin/export-data "https://data.4dnucleome.org/search/?type=StaticSection&limit=all" -u ACCESS_KEY_ID -p ACCESS_KEY_SECRET > new_static_section_inserts_file.json

- For images which desire to host externally (e.g. outside of repository or third-party URL), then it is suggested to upload images into a relevanet *sub-folder* (perhaps create an "/images/" folder for auxiliary images) of the **"4dn-dcic-public"** public S3 bucket. This bucket could also be used to host Markdown (.md) or other files, probably in the "/static-pages/" sub-folder, the URL of which can be used in the "file" field of StaticSections (will require a PATCH to Page or StaticSection to update 4DN Item content from file).

Permissions

Currently may set a `status` of "draft", "published", or "deleted" for any Page or StaticSection and permissions will work accordingly. Permissions by lab/user should work in same way as for other Items, but this hasn't yet been tested.

StaticSections Above Search Results

5.2.12.4 Simplification & Future < THIS WILL SUPERCEDE SYSINFOS MAPPING >

If we like this structure of having a static page or block for (almost) each `@type`, we could simplify greatly by getting rid of the Sysinfo Item & just having search.py look-up if any page w/ name `'/search-info-header/' + @type` exists and then including its contents into a 'search_header_content' property as part of search results/response JSON.

5.2.12.5 BELOW SYSINFOS APPROACH WILL BE DEPRECATED SOON BUT FOR NOW STILL FUNCTIONAL

5.2.12.6 Static Section Header `@type` Mapping

Currently this can be dynamically updated via the `SysInfo` Item : `/sysinfos/search-header-mappings/`

The Item `/sysinfos/search-header-mappings/` must exist in database for any static content to appear. Else will get nothing in area above search results. SysInfo cannot be inserted via deploy and must be POSTed in.

Do this on any instances we want mappings: https://gyazo.com/de6758e68ca898101218ad3d95687569 , with "mapping" taking the correct form (PATCHing subsequently after creation for updates).

Again, the name of the sysinfo object **MUST** be **`search-header-mappings`**

POST to `<host>/sysinfo/` :

```
{
  "name" : "search-header-mappings",
  "title" : "Search Header Mapping",
  "description": "Mapping of Static search result header URIs to Item @type",
```

(continues on next page)

(continued from previous page)

```
  "mapping" : {
      "WorkflowRun" : "/static-sections/search-info-header.WorkflowRun",
      "Workflow" : "/static-sections/search-info-header.Workflow"
  }
}
```

PATCH to `<host>/sysinfo/search-header-mappings`:

```
{
  "mapping" : {
      "WorkflowRun" : "/static-sections/search-info-header.WorkflowRun",
      "Workflow" : "/static-sections/search-info-header.Workflow",
      "FileSetMicroscopeQc" : "/static-sections/search-info-header.
↪FileSetMicroscopeQc"
  }
}
```

The "value" in the 'mapping' dictionary/object is the @id or link to a StaticSection Item. Here these static sections are referenced by their name (rather than UUID). In order to allow such a link to your StaticSection, ensure the 'name' of it doesn't have any slashes (`/`) or hashes (`#`). For example, in the case above the names are `search-info-header.WorkflowRun`, `search-info-header.Workflow`, & `search-info-header.FileSetMicroscopeQc`.

Auto-Generated Help Dropdown Menu

Pages have an optional `children` field which holds an array of other Pages (as linkTos). Routes of child pages **MUST** extend the parent route. For example, page with `name` == "help/submitter-guide" must have children with `names` in the form of "help/submitter-guide/something". The (sub-)children of the top level "help" page are automatically added to the top Help menu dropdown.

5.2.13 Reverse links

Reverse (rev) links are actually a pretty cool thing. Any time you have one object link to another object, through a calculated or schema based property, a rev link allows you to easily create the reverse direction link on the object that was being linked to. In addition, rev links take the status of the item that we are reverse linking into account – we do not want to create rev links to items that have a status of 'deleted', for example. In ENCODE, rev links were represented by linkFrom connections. We have changed that to only use linkTo.

Here is a simple example for the experiment item type (src/types/experiment.py):

Experiment sets have a linkTo experiments through the array *experiments_in_set* field. To make a rev link back from the experiment to the experiment sets, we must define the rev link and then create a calculated property that populates the linkTo.

For the first part, we define the *rev* property on the Experiment object. It is a dictionary that is keyed by the rev link name and has a value of *(<item type to rev link>, <field to rev link>)*. It would be defined as such:

‘‘‘

rev = { 'experiment_sets': ('ExperimentSet', 'experiments_in_set'),

}

‘‘‘

You can read this is as: we want to create a reverse link to ExperimentSet using the *experiments_in_set* field. Next, we will define a calculated property on the Experiment that will call this *rev* and create a list of actual linkTos.

‘‘‘ @calculated_property(schema={

“title”: “Experiment Sets”, “description”: “Experiment Sets to which this experiment belongs.”, “type”: “array”, “exclude_from”: [“submit4dn”, “FFedit-create”], “items”: {

“title”: “Experiment Set”, “type”: [“string”, “object”], “linkTo”: “ExperimentSet”

}

}) def experiment_sets(self, request):

return self.rev_link_atids(request, “experiment_sets”)

‘‘‘

That’s pretty much it! Now you have an automatic rev link that will be created on your experiment back to your experiment set. To embed values from the experiment set, you can add them to your *embedded_list* like any other object. For example, to embed the accession of the experiment set, you would add:

‘‘‘ embedded_list += [

‘experiment_sets.accession’

5.2.13.1]

There are a couple things going on behind the scenes that we should be aware of. Both are defined on the base Item class (src/types/base.py). First, we have a method called *rev_link_atids* on Item that MUST be called within your calculated property creating the rev links. It is actually responsible for generating the rev links from snovault and turning them from uuids to @ids. The code for the method is below (you should not need to change it)

‘‘‘ <a method for Item class>

def rev_link_atids(self, request, rev_name): “”” Returns the list of reverse linked items given a defined reverse link, which should be formatted like: rev = {

‘<reverse field name>’: (‘<reverse item class>’, ‘<reverse field to find>’),

}

“”” conn = request.registry[CONNECTION] return [request.resource_path(conn[uuid]) for uuid in

self.get_filtered_rev_links(request, rev_name)]

‘‘‘

Lastly, there is an attribute on Item called *filtered_rev_statuses*. It has a tuple value and serves to filter out all of the items of the given statuses from your rev links. This is crucial to the rev links working – we do not want to rev link to items with ‘deleted’ or ‘replaced’ statuses. This attribute may be overloaded on any item type to provide more fine-grained filtering. In base.py, it is:

```
` filtered_rev_statuses = ('deleted', 'replaced') `
```

In snovault, check out src/snovault/resources.py for the underlying *get_filtered_rev_links* and *get_rev_links* functions that provide the foundation for *rev_link_atids*.

5.2.14 UNIT Testing

5.2.14.1 Python : what & where

- `test_schema` : testing if mixins load from schema, and schema sytanctically correct
- `test_type_<object>` : test type sepcific stuff, minux embedding, calculated properties, update, etc..
- `test_search` : test effects of embedding and what not on search

5.2.14.2 JavaScript

Unit tests in JavaScript are performed with **Jest**, and initialized via `npm test <testfilenameprefix>` where testfilenameprefix is the first part (before `.js`) of the filename located in `src/encoded/static/components/__tests__`. Run `npm test` without arguments to run all tests. Execution of all tests is also automatically triggered in Travis upon committing or pull requesting to the GitHub repository.

Guidelines

- Look at current tests to get understanding of how they work.
- Check out the **Jest** API.
- Check out the React **TestUtils** documentation.
- If you need to test AJAX calls, utilize **Sinon** to create a **fake server** inside testing scripts, which will also patch XMLHttpRequest to work within tests. For example, in a `.../__tests__/` file, can have something resembling the following: ‘‘‘javascript sinon = require(‘sinon’); var server = sinon.fakeServer.create();

// Setup dummy server response(s) server.respondWith(

“PATCH”, // Method context[‘@id’], // Endpoint / URL [

200, // Status code { “Content-Type” : “application/json” }, // Headers ‘{ “status” : “success” }’ // Raw data returned

]

);

// Body of test doSomeFunctionsHereWhichSendAJAXCalls(); // Any code with AJAX/XHR calls. server.respond(); // Respond to any AJAX requests currently in queue. expect(myNewValue).toBe(whatMyNewValueShouldBe); // Assert state in Jest that may have changed in response to or after AJAX call completion.

doSomeMoreFunctionsWithAJAX(); server.respond(); expect(myOtherNewValue).toBe(whatMyOtherNewValueShouldBe);

server.restore(); // When done, restore/unpatch the XMLHttpRequest object. ‘‘‘

5.2.15 Load Testing with Locust

Locust is a load testing tool that can easily be provisioned to run against any of our environments. There are two required files - `config.json` and `<env>.json` where <env> is the environment you’d like to run load testing on.

5.2.15.1 Supported Environments

- Data
- Staging
- Mastertest
- Hotseat

5.2.15.2 Config.json

To configure the load testing you must write the file `config.json`. A basic one is provided as a default and updates are gitignore'd (so if a change is needed you will have to force it). There are two important fields - `routes` and `envs`. `envs` specifies a mapping from environment name to the associated URL. This should never really change unless our URL's do. You could also add new environments here, but a seperate key file is necessary. The second field is `routes`. This is where you specify what routes you'd like Locust to hit. Locust will hit all routes specified approximately evenly. This behavior can be changed by explicitly specifying your routes in `ff_locust.py`.

5.2.15.3 <env>.json

In this file you will need to add your access keys for the environment you'd like to test. By default we will try to locate your credentials in <env>.json. If we do not locate them the program will exit. Provide the field 'username' with your access key ID and 'password' with the secret. You can generate new access keys from your user page when accessing the relevant portal. These keys will be different across environments, hence the need to provide a separate `<env>.json` file per environment you'd like to test

5.2.15.4 Command Line Arguments

usage: main.py [-h] [–time TIME] [–nclients NCLIENTS] [–rate RATE] [–lower LOWER] [–upper UPPER] config key

Locust Load Testing

positional arguments: config path to config.json key path to <env>.json

optional arguments:

-h, --help	show this help message and exit
--time TIME	time to run test for, default 1m. Format: 10s, 5m, 1h, 1h30m etc.
--nclients NCLIENTS	number of clients, default 10
--rate RATE	number of clients to hatch per second, default 10
--lower LOWER	lower bound on time to wait between requests, default 1
--upper UPPER	upper bound on time to wait between requests, default 2

5.2.16 Introduction for Users

- The 4DN Data Portal will be the primary access point to the omics and imaging data, analysis tools, and integrative models generated and utilized by the 4DN Network.

- The primary high level organizing principle for the data is sets of replicate experiments.
- A good entry point for exploring available data is the Browse Page.
- See *below* for an overview of our metadata model.
- As of September 2017, the portal is currently open to the network for data submission for standard functional genomics experiments (Hi-C and variants, ChIA-PET and variants, RNA-seq, ChIP-seq, ATAC-seq).
- Continuing developments in the metadata model and data portal are ongoing.

5.2.16.1 Notes for prospective submitters

If you would like submit data to the portal:

- You will need to create a user account.
- Please skim through the *metadata structure*.
- Check out the other pages in the **Help** menu for detailed information on the submission process.
- Of note are the required metadata for the biological samples used in experiments, which is specified on this page.
- We can set up a webex call to discuss the details and the most convenient approach for your existing system.

Metadata Structure

The DCIC, with input from different 4DN Network Working groups, has defined a metadata structure to describe:

- biological samples
- experimental methods
- data files
- analysis steps
- and other pertinent data.

The framework for the the metadata structure is based on the work of ENCODE DCC.

The metadata is organized as objects that are related with each other. An overview of the major objects is provided in the following slides.

In our database:

- The objects are stored in the JavaScript Object Notation format.
- The schemas for the different object types are described in JSON-LD format.
- The json schemas can be found here.
- A documentation of the metadata schema is also available as a google doc here.

5.2.17 Getting Started (User)

5.2.17.1 Overview

In order to make your data accessible, searchable and assessable you should submit as much metadata as possible to the 4DN system along with the raw files you have generated in your experiments.

These pages are designed to

- show you how to find out what kind of metadata we collect for your particular type of experiment
- introduce the mechanisms by which you can submit your metadata and data to the 4DN data portal.

For an overview of the metadata structure and relationships between different items please see the slides available on the metadata introductory page.

We have three primary ways that you can submit data to the 4DN data portal.

Web Submission

The online web submission forms are best used

- To submit one or a few experiments.
- To edit one or a few fields of an already submitted but not yet released item.
- As a hands on way to gain familiarity with the 4DN data model.

Documentation on how to get started with this interface is here.

Data Submission via Spreadsheet

The excel metadata workbooks

- Are useful for submitting metadata and data for several experiments or biosamples
- Can be used to make bulk edits of submitted but not yet released metadata
- Contain multiple sheets where each sheet corresponds to an object type and each column a field of metadata
- Can be generated using the Submit4DN software
- Are used as input to the Submit4DN software which validates submissions and pushes the content of the forms to our database.

Documentation of the data submission process using these forms can be found here.

REST API

For both meta/data submission and retrival, you can also access our database directly via the REST-API.

- Data objects exchanged with the server conform to the standard JavaScript Object Notation (JSON) format.
- Our implementation is analagous to the one developed by the ENCODE DCC.

If you would like to directly interact with the REST API for data submission see the documentation here.

5.2.17.2 Notes on Experiments and Replicate Sets

Biological replicates

- The 4DN Consortium strongly encourages that experiments be performed using at least two different preparations of the same source biomaterial - i.e. bioreplicates.
- When submitting metadata you should submit two Experiments that use the same Biosource, but have different Biosamples.
- In many cases the only difference between Biosamples may be the dates at which the cell culture or tissue was harvested.
- The experimental techniques and parameters will be shared by all experiments of the same bioreplicate set.

Technical replicates

- Multiple sequencing runs performed at different times using a library prepared from the same Biosample and the same methods up until the sample is sent to the sequencer - i.e. technical replicates.

Submitting replicate information

- The replicate information is stored and represented as a set of experiments that includes labels indicating the replicate type and replicate number of each experiment in the set.
- The mechanism that you use to submit your metadata will dictate the type of item that you will associate replicate information with
  - In excel workbooks bioreplicate and technical replicate numbers are entered in the Experiment sheet
  - Using the API you directly associate the replicate information (*i.e. replicate number and the experiment identifier*) with the ExperimentSetReplicate objects.
  - Using the web submission interface the replicate numbers and linked experiments are added from the ExperimentSetReplicate page
- In the database the information will always end up directly associated with ExperimentSetReplicate objects.
- Specific details on formatting information regarding replicates is given in the Spreadsheet Submission page.
- When submitting using the REST API you should format your json according to the specifications in the schema as described in the REST API page.

5.2.17.3 Referencing existing objects

Using aliases

Aliases are a convenient way for you to refer to other items that you are submitting or have submitted in the past.

- An alias is a lab specific identifier that you can assign to any item
- An alias takes the form of *lab:id_string* eg. `parklab:myalias`.
- An alias must be unique within all items.
- Generally it is good practice to assign an alias to any item that you submit

- If you use the Online Submission Interface to create new items designating an alias is the first required step.
- Once you submit an alias for an Item then that alias can be used as an identifier for that Item in the current submission as well as in any subsequent submission.

Other ways to reference existing items

You don't need to use an alias if you are referencing an item that already exists in the database.

Any of the following can be used to reference an existing item in an excel sheet or when using the REST-API.

- **accession** - Objects of some types (eg. Files, Experiments, Biosamples, Biosources, Individuals. . .) are *accessioned*, e.g. 4DNEX4723419.
- **uuid** - Every item in our database is assigned a "uuid" upon its creation, e.g. "44d3cdd1-a842-408e-9a60-7afadca11575".
- **type/id** in a few cases object specific identifying terms are also available, eg. award number for awards, or lab name for labs. (see table below)

Object	Field	type/ID	ID
Lab	name	/labs/peter-park-lab/	peter-park-lab
Award	number	/awards/ODO1234567-01/	ODO1234567-01
User	email	/users/test@test.com/	test@test.com
Vendor	name	/vendors/fermentas/	fermentas
Enzyme	name	/enzymes/HindIII/	HindIII
Construct	name	/constructs/GFP-H1B/	GFP-H1B

- Many of the objects that you may need for your submissions may already exist on the 4DN web site.
- We encourage submitters to use existing database items as much as possible.
- Common reusable items include:
  - Vendors
  - Enzymes
  - Biosources
  - Protocols
- For example, if there is an existing biosource (e.g. accession 4DNSRV3SKQ8M for H1-hESC (Tier 1)) for the new biosample you are creating, you should reference the existing one instead of creating a new one.

5.2.17.4 Getting Added as a 4DN User or Submitter

Before you can view protected lab or project data or submit data to the 4DN system you must be a registered user of the site and have the appropriate access credentials.

- To view lab data that is still in the review phase you must be registered as a member of the lab that produced the data.
- To submit metadata and files you must be designated as a submitter for a lab
- Most current 4DN lab members should already be registered in our system.

For instructions on creating an account, please see this page.

Metadata and data accessibility.

- Most metadata items have the following default permissions:
  - members of the submitting lab can view
  - submitters for the lab can edit
  - to help you review and edit a lab's submissions the DCIC data wranglers can view and edit
- Once the data and metadata are complete and quality controlled, they will be released according to the data release policy adopted by the 4DN network.
- After release the data can no longer be edited by data submitters - contact the DCIC to report data issues and we can work together to get them resolved

5.2.17.5 Getting Connection Keys for the 4DN-DCIC servers

If you have been designated as a submitter for the project and plan to use either our spreadsheet-based submission system or the REST-API an access key and a secret key are required to establish a connection to the 4DN database and to fetch, upload (post), or change (patch) data. Please follow these steps to get your keys.

1. Log in to the 4DN website with your username (email) and password. If you have not yet created an account, see this page for instructions.
2. Once logged in, go to your "Profile" page by clicking **Account** on the upper right side of the page.
3. In your profile page, click the green "Add Access Key" button, and copy the "access key ID" and "secret access key" values from the pop-up page. *Note that once the pop-up page disappears you will not be able to see the secret access key value.* However, if you forget or lose your secret key you can always delete and add new access keys from your profile page at any time.
4. Create a file to store this information.
   - The default parameters used by the submission software is to look for a file named "key-pairs.json" in your home directory.
   - However you can specify your own filename and file location as parameters to the software (see below).
   - The key information is stored in json format and is used to establish a secure connection.
   - the json must be formatted as shown below - replace key and secret with your new "Access Key ID" and "Secret Access Key".
   - You can use the same key and secret to use the 4DN REST-API.

Sample content for keypairs.json

```
{
  "default": {
    "key": "ABCDEFG",
    "secret": "abcdefabcd1ab",
    "server": "https://data.4dnucleome.org/"
  }
}
```

Tip: If you don't want to use that filename or keep the file in your home directory you can use:

- the `--keyfile` parameter as an argument to any of the scripts to provide the path to your keypairs file.
- the `--key` parameter to indicate a stored key name.

```
import_data --keyfile Path/name_of_file.json --key
NotDefault
```

5.2.18 Account Creation

5.2.18.1 If you are a data submitter for a 4DN lab or are new to the project

- Please email data wranglers at support@4dnucleome.org to get set up with an account with the access credentials for your role.
- Please provide an email address which you wish to use for your account and CC your PI for validation purposes. **The email associated with the account you use for login must be the same as the one registered with the 4DN-DCIC.**
  - This can be any email address (*e.g. an institutional email account*) but must be connected to either a Google or Github account.
  - For more information on linking your institutional email account to a Google account, see below.

5.2.18.2 Signing in with your institutional email address

- The DCIC uses the OAuth authentication system which will allow you to login with a Google or GitHub account.
- If you prefer to use your institutional email address to log in to the portal (recommended), you need to have a Google or GitHub account registered with that email address.
- If you do not already have a Google or GitHub account with your email address, you can set up one up by visiting the Google account creation page with the non-gmail option.

NOTE that it is important not to register this account to have gmail as your institutional email address must be the primary email associated with the google account for authentication to work properly!

Once your account request is processed, you will then be able to log in with the 'LOG IN WITH GOOGLE' option using your institutional email address and Google account password.

5.2.19 Overview

- The 4DN consortium will collect metadata on the preparation of a biological sample (biosample) in order to make the data FAIR, Findable, Accessible, Interoperable and Reusable, to the extent that such a service benefits the network and scientific community at large.
- Many 4DN experiments are performed using cell lines. Other experiments may be performed on isolated primary cells or tissues.

- Experimenters may also perform assays where cells are transiently treated, for example by addition of a drug or introduction of a silencing construct, or stably genomically modified through Crispr technologies.

This page outlines and describes the types of metadata that is requested for biological samples.

- The first part of the document outlines the few fields shared by all biosamples.
- The Cell Lines and Samples Working Group has focused on developing requirements for cell line metadata and this is the primary focus of the *remainder of this document*.

Note that the working group is still discussing some of the metadata and requirements are evolving. If you have any questions or concerns please feel free to 'contact us <mailto:support@4dnucleome.org>'_.

5.2.20 Basic Biosample Metadata

5.2.20.1 Biosample Fields

`description` - Required {:.text-400}

- A brief specific description of the biosample
- example "K562 cells prepared for in situ Hi-C experiments"
- example "GM12878 cells modified using Crispr to delete CTCF sites in the PARK2 region prepared for chromatin capture experiments"

`biosource` - Required {:.text-400}

- The value of this field is a reference to usually one **Biosource** object whose metadata is submitted separately.
- This **Biosource** object refers to a cell line, tissue or primary cell and has its own associated metadata.
  - **NOTE**: The tiered cell lines all have an existing biosource in the database that can be re-used and referenced by it's accession, alias or uuid - while other biosources may require you to submit metadata for them.
- It is possible, though rare, for a single biosample to consist of more than one biosource - eg. pooling of two different cell lines - in these cases you can reference multiple biosources in this field.

`cell_culture_details` - Required only for cultured cell lines {:.text-400}

- The value of this field is a reference to a *BiosampleCellCulture* object whose metadata is submitted separately and is detailed in the *Cell Culture Metadata section below*.

`modifications` - Required if cells have been genomically modified {:.text-400}

- **Genetic modifications** - this field is **required** when a Biosample has been genomically modified eg. Crispr modification of a cell line.
- The value of this field is a list of one or more references to a **Modification** object whose metadata is submitted separately.

- Modifications include information on expression or targeting vectors stably transfected to generate Crispr'ed or other genomically modified samples.

`treatments` - Required if cells have been treated 'transiently' with drugs or by transfection. {:.text-400}

- This field is used when a Biosample has been treated with a chemical/drug or transiently altered using RNA interference techniques.
- The value of this field is a reference to a **Treatment** object whose metadata is submitted separately.
- There are currently two general types of treatments - more will be added as needed.
  1. Addition of a drug or chemical
  2. Transient or inducible RNA interference

`biosample_protocols` - Optional {:.text-400}

- Protocols used in Biosample Preparation - this is distinct from SOPs and protocol for cell cultures.
- example protocol description "Preparation of isolated brain tissue from BALB/c adult mice for chromatin capture experiments"
- The value of this field is a list of references to a **Protocol** object - an alias or uuid.
- The **Protocol** object can include an attachment to a pdf document describing the steps of the preparation.
- The **Protocol** object is of type 'Biosample preparation protocol' and can be further classified as 'Tissue Preparation Methods' if applicable.

5.2.21 Cell Culture Metadata

- The consortium has designated 4 cell lines as Tier 1, which will be a primary focus of 4DN research and integrated analysis.
- A number of other lines that are expected to be used by multiple labs and have approved SOPs for maintaining them have been designated Tier 2.
- In addition, some labs may wish to submit datasets produced using other cell lines.

To maintain consistent data standards and in order to facilitate integrated analysis the Cell Lines and Samples Working Group has adopted the following policy.

Certain types of metadata, if not submitted will prevent your data from being flagged "gold standard". For your data to be considered "gold standard", you will need to obtain your cells from the approved source and grow them precisely according to the approved SOP and include the following required information:

1. A light microscopy image (DIC or phase contrast) of the cells at the time of harvesting (omics) or under each experimental condition (imaging);
2. culture start date, culture harvest date, culture duration, passage number and doubling number

Other metadata is strongly encouraged and the exact requirements may vary somewhat depending on the cell type and when the data was produced (i.e. some older experiments can be 'grandfathered' in even if they do not 'pass' all the requirements).

The biosample cell culture metadata fields that can be submitted are described below.

5.2.21.1 BiosampleCellCulture fields

`description` - Strongly Encouraged {:.text-400}

- A short description of the cell culture procedure
- example “Details on culturing a preparation of K562 cells”

`morphology_image` - Required {:.text-400}

- Phase Contrast or DIC Image of at least 50 cells showing morphology at the time of collection
- This is an authentication standard particularly relevant to Tiered cell lines.
- The value of this field is a reference to an **Image** object that needs to be submitted separately.

`culture_start_date` - Required {:.text-400}

- The date the the cells were most recently thawed and cultured for the submitted experiment
- Date can be submitted in as YYYY-MM-DD or YYYY-MM-DDTHH:MM:SSTZD ((TZD is the time zone designator; use Z to express time in UTC or for time expressed in local time add a time zone offset from UTC +HH:MM or -HH:MM).
- example Date only (most common use case) - “2017-01-01”
- example Date and Time (uncommonly used) -“2017-01-01T17:00:00+00:00” - note for time; hours, minutes, seconds and offset are required but may be 00 filled.

`culture_harvest_date` - Required {:.text-400}

- The date the culture was harvested for biosample preparation.
- Date format as above.

`culture_duration` - Required {:.text-400}

- Total Days in Culture.
- Total number of culturing days since receiving original vial, including pyramid stocking and expansion since thawing the working stock, through to harvest date.
- The field value is a number - can be floating point
- example “5”
- example “3.5”

`passage_number` - Required {:.text-400}

- Number of passages since receiving original vial, including pyramid stocking and expansion since thawing the working stock, through to harvest date.
- Only integer values are allowed in this field eg. 3, 5, 11

`doubling_number` - Required {:.text-400}

- The number of times the population has doubled since the time of thaw (culture start date) until harvest.
- This may be determined and reported in different ways
  1. passage ratio and number of passages
  2. direct cell counts.
- Therefore, this field takes a string value
- example "7.88"
- example "5 passages split 1:4"

`follows_sop` - Required {:.text-400}

- Flag to indicate if the 4DN SOP for the specified cell line was followed - options '**Yes**' or '**No**'
- If a cell line is not one of the 'Tiered' 4DN lines this field should be set to 'No'

`protocols_additional` - Required if 'follows_sop' is 'No' {:.text-400}

- Protocols used in Cell Culture when there is deviation from a 4DN approved SOP.
- Protocols describing non-4DN protocols or deviations from 4DN SOPs, including additional culture manipulations eg. stem cell differentiation or cell cycle synchronization if they do not follow recommended 4DN SOPs
- The value of this field is a list of references to a **Protocol** object - an alias or uuid.
- The **Protocol** object can include an attachment to the pdf document.

`doubling_time` - Optional {:.text-400}

- Population Doubling Time
- The average time from thaw (culture start date) until harvest it takes for the population to double.
- Researchers can record the number of times they split the cells and by what ratio as a simple approximation of doubling time. This is especially important for some cell lines eg. IMR90 (a mortal line) and HI and H9 human stem cells.
- eg. '2 days'

`authentication_protocols` - Optional {:.text-400}

- References to one or more **Protocol** objects can be submitted in this field.
- The **Protocol** objects should be of the type 'Authentication document'
- The **Protocol** object can be further classified by indicating a specific classification eg. 'Karyotyping authentication' or 'Differentiation authentication'.
- The **Protocol** description should include specific information on the kind of authentication

  - example “g-banding karyotype report”
  - example “images of FoxA2 and Sox17 expression in differentiated endoderm cells”
- The **Protocol** object can include an attachment to the pdf or image document.

`karyotype` - Optional description of cell ploidy and karyotype {:.text-400}

- Description of cell Ploidy - a textual description of the population ploidy and/or karyotype.
- Important for potentially genomically unstable lines and strongly encouraged if the passage number of an unstable line is greater than 10.
- A textual description of chromosome count and any noted rearrangements or copy number variations.
- examples include
  - chromosome counts or structural variation using sequencing data
  - chromosome counts using droplet PCR
  - cytological G-banding
- Using this field allows this information to be queried in searches.
- **NOTE** An image or authentication document (see above) may be submitted in place or in addition to this.

`differentiation_state` - Optional {:.text-400}

- For cells that have undergone differentiation a description of the differention state and markers used to determine the state.
- Using this field allows this information to be queried in searches.
- example ‘Definitive endoderm as determined by the expression of Sox17 and FoxA2’
- **NOTE** An authentication document (see above) can be submitted in place or in addition to this.

`synchronization_stage` - Optional {:.text-400}

- If a culture is synchronized then the cell cycle stage or description of the point from which the biosample used in an experiment is prepared.
- Using this field allows this information to be queried in searches.
- example ‘M-phase metaphase arrested cells’
- **NOTE** An authentication document (see above) can be submitted in place or in addition to this.

`cell_line_lot_number` - Strongly Suggested for non-Tier 1 cells {:.text-400}

- For 4DN Tier2 or unclassified cell lines - a lot number or other information to uniquely identify the source/lot of the cells

5.2.22 Excel Submission

5.2.22.1 Overview

- Metadata and data can be submitted to our platform using Microsoft Excel WorkBooks that describe related items in separate sheets.
- This section provides detailed information on how to use the WorkBooks.
- You can check out the example WorkBook we prepared for the data from Rao et. al. 2014 to familiarize yourself with the general structure.
- Based on the type of experiment(s) for which you plan to submit data, the data wranglers can provide you with an Excel WorkBook containing several WorkSheets.
- Each sheet corresponds to an Item type in our metadata database.
- The workbook provided should contain all the sheets that you may need for your submission.
- You can refer to *this table* for information on all the Item types available in the database.
- Each sheet should also contain all the data fields that can be submitted for that Item type.
- Depending on if you have submitted data before or if you are using shared reagents that have been submitted by other labs, you may not need to provide information on every sheet or in every field.

Organization of the Workbook

- Generally, it makes sense to begin with the left most sheet in the workbook as the sheets in a workbook are ordered so that Items that have fields that take a reference to another Item as their value appear 'after' i.e. to the right of that Item's sheet in the workbook.
- A sheet for an Item starts with a row of field names.
- *Absolutely required fields are marked with a leading asterisk (eg. *experiment_type).* - failure to supply a value in these fields will cause an error
- The second row of the sheet indicates the type of the information expected for the fields.
- The third row includes a description of each of the fields.
- In some cases the values that you can submit for a particular field are constrained to a specific set of terms and when this is the case the possible values are shown in the fourth row.
- Any row that starts with "#" in the first column will be ignored, so you can add non-data rows for your own use.
- However, **PLEASE NOTE THAT THE FIRST 2 ROWS OF A SHEET SHOULD NOT BE MODIFIED.**

Excel Headers

1. Field name
2. Field type (string, number, array, embedded object)
3. Description
4. Additional info (comments and choices for fields with controlled vocabulary)

- You may notice that in some sheets there are additional commented rows that contain data values.
- These are rows corresponding to items that already exist in the database and can provide you with identifiers that you can reuse in your submission (see Referencing existing items).

- Only those items that either are associated with your lab or are already released to the public will appear in these commented data rows.
- Your data entry should begin at the first non-commented row.

5.2.22.2 Preparing Excel Workbooks

- A field can be one of a few different types;
  - string
  - number/integer
  - array/list
  - Item
- The type will be indicated in the second row.
- Most field values are strings:
  - a term from a controlled vocabulary, i.e. from a constrained list of choices
  - a string that identifies an Item
  - a text description.
  - If the field type is an array, you may enter multiple values separated by commas.

Basic field formats

Basic fields example

- There are some fields values that require specific formatting. These cases and how to identify them are described below.

In some cases a field value must be formatted in a certain way or the Item will fail validation. In most cases tips on formatting requirements will be included in the *Additional Info* row of the spreadsheet.

Examples of these are

- *Date* fields - YYYY-MM-DD format.
- *URLs* -checked for proper URI syntax.
- *patterns* - checked against simple regular expressions (eg. a DNA sequence can only contain A, T, G, C or N).
- *Database Cross Reference (DBxref) fields* that contain identifiers that refer to external databases

  - In many cases the values of these fields need to be in database_name:ID format. eg. an SRA experiment identifier 'SRA:SRX1234567' (see also *Basic fields example* above).
  - In a few cases where the field takes only identifiers for one specific databases the ID alone can be entered - eg. *'targeted_genes'* field of the Target Item enter gene symbols eg. PARK2, DLG1.

- Some fields in a Sheet for an Item may contain references to another Item.
- The referenced Item may be of the same or different type.
- Examples of this type of field include the *'biosource'* field in Biosample or the *'files'* field in the ExperimentHiC.
- The *'files'* field is also an example of a list field that can take multiple values.
- You can reference an item in the excel workbooks using one of four possible ways:
  1. lab-specific alias
  2. accession
  3. item-type-specific identifier
  4. UUID

More information about these four identifiers is provided in Using aliases.

- Some Items can contain embedded sub-objects that are stored under a single Item field name but that contain multiple sub-fields that remain grouped together.
- These are indicated in the Item spreadsheet using a '.' (dot) notation.

  For example the *"experiment_relations"* field has 2 sub-fields called *"relationship_type"*, and *"experiment"*. In the spreadsheet field names you will see *experiment_relations.relationship_type* and *experiment_relations.experiment*.

- If the Item field is designed to store a list of embedded sub-objects, you can enter multiple sub-objects by manually creating new columns and appending incremented integers to the fields names for each new sub-object.

  For example, to submit a total of three related experiments to an ExperimentHiC Item you would find the *experiment_relations.relationship_type* and *experiment_relations.experiment* columns, copy them and have total of 6 columns named:

and enter a valid *relationship_type* term and *experiment* identifier to each of the three pairs of columns.

Multiple linked columns for lists of embedded objects

static/img/docs/submitting-metadata/embedded_objects.png

- Ways that you can reference items that already exist in the 4DN database in your spreadsheet submission is described here.
- In some cases information for existing items will be present in the Excel Work Sheets provided for your submission.
- You can also check the existing items from *collection* pages that list all of them.

- The links for item lists can be constructed by `https://data.4dnucleome.org/ + plural-object-name` (e.g. https://data.4dnucleome.org/biosamples/) and the identifiers that can be used for collections are referenced in this table.

To submit supplementary metadata files, such as pdfs or images, use the **Image** or **Document** schemas, and include the path of the files in the **attachment** column. The path should be the full path to the supplementary file.

- All experiments must be part of a replicate set - even if it is a set containing only a single experiment.
- When preparing your submission you should determine how many replicate sets you will be submitting and create an entry - with an alias and preferably an informative description - for each set in the ExperimentSetReplicate sheet.

- Then when entering information about individual experiments on the specific **Experiment_** sheet you should:
  1. enter the alias for the replicate set to which the experiment belongs
  2. indicate the bioreplicate and technical replicate number for that experiment.
- In the example below the replicate set consists of five experiments categorized into one of two bioreplicates - bio_rep_no 1 and bio_rep_no 2, each of which contains three and two technical replicates, respectively.

5.2.22.3 Submitting Excel Workbooks

- The 4DN DCIC website has an REST API for fetching and submitting data.
- In our **Submit4DN** package the `import_data` script utilizes an organized bundle of REST API commands that parse the Excel workbook and submit the metadata to the database for you.
- The `get_field_info` script that is also part of the package can be used to generate the Excel workbook templates used for submission for all or a selected set of worksheets.
- The package can be installed from pypi.

The Submit4DN package is registered with Pypi so installation is as simple as:

```
pip3 install submit4dn
```

If it is already installed upgrade to the latest version:

```
pip3 install submit4dn --upgrade
```

The source code for the submission scripts is available on github.

Note if you are attempting to run the scripts in the wranglertools directory without installing the package, then in order to get the correct sys.path you need to run the scripts from the parent directory as modules using the -m flag.

```
python3 -m wranglertools.import_data  filename.xls
```

- You can use `import_data` either to upload new items or to modify metadata fields of existing items.
- This script will accept the excel workbook you prepared, and will upload every new item in the sheets.
- This script is also used to upload data files to the 4DN data store - this is done in a separate step after your File metadata has been successfully uploaded.

You will need to generate access keys to submit data. How to get these is described here.

- Before actually updating the 4DN database you can check your spreadsheet for formatting and missing required data by doing a 'dry run'.
- When you run the import_data script on your metadata excel workbook without the `--update` or `--patchall` arguments the system will test your data for compatibility with our metadata structure and report back to you any problems.
- The metadata will not be submitted to the database, so you can take advantage of this feature to test your excel workbook.

```
import_data My_metadata.xls
```

- When you submit your metadata, if a row in any sheet corresponds to a new item that has not previously been submitted to the 4DN database you will be POSTing that data via the REST API.
- Most of your entries in the first submission will be POSTs. To activate posting you need to include the `--update` argument to `import_data`.

```
import_data My_metadata.xls --update
```

- If you need to modify an existing item, you can use the patch function.
- To be able to match your item to the one on the server, a pre-existing identifier must be used in the spreadsheet.
- If you included an alias when you posted the item, you can use this alias to reference the existing item in the database – uuids, @ids, or accessions can also be used to reference existing items in the database.
- If you don't use the `--patchall` argument when you run `import_data` and an existing entry is encountered, the script will prompt you 'Do you wish to PATCH this item?'. You will be prompted for every existing item that is found in your workbook.
- The `--patchall` argument will allow automatic patching of each existing item, bypassing the prompts.

```
import_data My_metadata.xls --patchall
```

- If for some reason the script fails in the middle of the upload process or errors are encountered for certain items, some items will have been posted while others will have not.
- When you fix the problem that caused the process to terminate, you can rerun the script using both the `--patchall` and `--update` arguments.

- Those items that had already been posted will be 'patched' using the data in the sheet and the items that had not been posted yet will be loaded.

```
import_data My_metadata.xls --patchall --update
```

- Functionality that will allow the deletion of all the data in a single field of an existing Item exists - however this can be a potentially dangerous operation. If you determine that you need this functionality please contact us at the DCIC for more information.
- The 4DN databased distinguishes two main categories of files:
  1. files that support the metadata, such as Documents or Images
  2. data files for which metadata is gathered and are specified in specific File items/sheets (eg. FileFastq).
- The first category can be uploaded along with the metadata by using the "attachment" fields in the excel workbook (eg. pdf, png, doc, ...) as *described previously*.
- The second category includes the data files that are the results of experiments, eg. fastq files from HiC experiments.
  - These data files are bound to a File item with a specific type eg, FileFastq that contains relevant metadata about the specific result file.
  - Metadata for a file should be submitted as part of your experiment metadata submission as described above.
  - The actual file upload to the 4DN file store in the cloud will happen in a subsequent submission step. **NOTE that the filename is not part of the initial File metadata submission.**
  - This second step will be triggered by a successful metadata submission that passes review by the 4DN DCIC.

To upload your files:

1. use the file submission excel sheet provided
2. copy paste all your file (FileFastq) aliases from your metadata excel sheet to the aliases field of the file submission sheet
3. Under filename enter the full paths to your files
4. use import_data with the `--patchall` argument to start upload.

The DCIC automatically checks file md5sums to confirm successful upload and to ensure that there are no duplicate files in the database.

Tip Upload using ftp is also supported, however the process currently transfers the files to your hard drive, uploads them to our system, and then deletes the copy from your local hard drive. The files are processed sequentially so you need to have at least the amount of free space on your hard drive as the size of the largest file you wish to upload. In addition, you must include your ftp login credentials in the ftp url, **which is definitely not a security best practice**. For these reasons, if at all possible, it is recommended to install the Submit4DN package onto the server hosting the files to be submitted and use import_data as described above. However, if that is not an option then your ftp urls should be formatted as follows:

`ftp://username:password@hostname/path/to/filename`

To replace a file that has already been uploaded to 4DN - that is to associate a different file with existing metadata,

- in the filename field include the new path for the existing alias

- **NOTE that every time you patch with a filename (even if it is the same filename) the file will be uploaded. Please use care when including a filename in your File metadata to avoid unnecessary uploads.**
- We plan to avoid this issue in future releases by pre-checking md5sums.

5.2.22.4 Generate a new Template Workbook

To create the data submission xls forms, you can use get_field_info, which is part of the Submit4DN package.

The scripts accepts the following parameters:.

Examples generating a single sheet:

To get the complete list of relevant sheets in one workbook:

5.2.23 Schema information

Schema Filename	Worksheet Name	Collection Name(s)
analysis_step.json	AnalysisStep	analysis-steps, analysis_step
award.json	Award	award(s)
biosample.json	Biosample	biosample(s)
biosample_cell_culture.json	BiosampleCellCulture	biosample-cell-cultures, biosample_cell_culture
biosource.json	Biosource	biosource(s)
construct.json	Construct	construct(s)
document.json	Document	document(s)
enzyme.json	Enzyme	enzyme(s)
experiment_atacseq.json	ExperimentAtacseq	experiments-atacseq, experiment_atacseq
experiment_capture_c.json	ExperimentCaptureC	experiments-capture-c, experiment_capture_c
experiment_chiapet.json	ExperimentChiapet	experiments-chiapet, experiment_chiapet
experiment_hi_c.json	ExperimentHiC	experiments-hi-c, experiment_hi_c
experiment_mic.json	ExperimentMic	experiments-mic, experiment_mic
experiment_repliseq.json	ExperimentRepliseq	experiments-repliseq, experiment_repliseq
experiment_seq.json	ExperimentSeq	experiments-seq, experiment_seq
experiment_set.json	ExperimentSet	experiment-sets, experiment_set
experiment_set_replicate.json	ExperimentSetReplicate	experiment-set-replicates, experiment_set_replicate
file_calibration.json	FileCalibration	files-calibration, file_calibration
file_fastq.json	FileFastq	files-fastq, file_fastq
file_processed.json	FileProcessed	files-processed, file_processed
file_reference.json	FileReference	files-reference, file_reference
file_set.json	FileSet	file-sets, file_set
file_set_calibration.json	FileSetCalibration	file-set-calibrations, file_set_calibration
genomic_region.json	GenomicRegion	genomic-regions, genomic_region
image.json	Image	image(s)
imaging_path.json	ImagingPath	imaging-paths, imaging_path
individual_human.json	IndividualHuman	individuals-human, individual_human
individual_mouse.json	IndividualMouse	individuals-mouse, individual_mouse
lab.json	Lab	lab(s)
modification.json	Modification	modification(s)
ontology.json	Ontology	ontology(s)

Continued on next page

Table 1 – continued from previous page

Schema Filename	Worksheet Name	Collection Name(s)
ontology_term.json	OntologyTerm	ontology-terms, ontology_term
organism.json	Organism	organism(s)
protocol.json	Protocol	protocol(s)
publication.json	Publication	publication(s)
publication_tracking.json	PublicationTracking	publication-trackings, publication_tracking
quality_metric_bamqc.json	QualityMetricBamqc	quality-metrics-bamqc, quality_metric_bamqc
quality_metric_fastqc.json	QualityMetricFastqc	quality-metrics-fastqc, quality_metric_fastqc
quality_metric_flag.json	QualityMetricFlag	quality-metric-flags, quality_metric_flag
quality_metric_pairsqc.json	QualityMetricPairsqc	quality-metrics-pairsqc, quality_metric_pairsqc
software.json	Software	software(s)
sop_map.json	SopMap	sop-maps, sop_map
summary_statistic.json	SummaryStatistic	summary-statistics, summary_statistic
summary_statistic_hi_c.json	SummaryStatisticHiC	summary-statistics-hi-c, summary_statistic_hi_c
target.json	Target	target(s)
treatment_agent.json	TreatmentAgent	treatments-agent, treatment_agent
treatment_rnai.json	TreatmentRnai	treatments-rnai, treatment_rnai
user.json	User	user(s)
vendor.json	Vendor	vendor(s)
workflow.json	Workflow	workflow(s)
workflow_mapping.json	WorkflowMapping	workflow-mappings, workflow_mapping
workflow_run.json	WorkflowRun	workflow-runs, workflow_run
workflow_run_sbg.json	WorkflowRunSbg	workflow-runs-sbg, workflow_run_sbg

5.2.24 Web Submission

- An online submission interface has been developed to help with the submission of 4DN metadata.
- This web interface is especially useful for;
  - submitting one or a few experiments
  - editing the metadata for an existing experiment
  - understanding object dependencies in our metadata schemas (for example learning that every experiment needs a type and a biosample).
- The system has been developed as a submission wizard that allows both the stepwise creation of database objects and full submission of an entire experiment with all required associated objects.
- We do recommend you review the information on the Getting Started page to get some tips on important concepts like *aliases* and *Replicate Sets*.

5.2.24.1 Creating New Items

There are several possible 'entry' points to a web submission

- You may want to start by entering metadata for an ExperimentSetReplicate object or an Experiment object of a particular type (eg. a Hi-C experiment or Microscopy experiment).
- You can start by creating experiments and then as a subsequent step associating multiple experiments with a Replicate Set.
- You can start your submission at a lower level item type (eg. Biosample) if that makes things easier for you.

To create a new item

1. Navigate to an item of the type for which you want to create metadata.
2. You can find **Create** and **Edit** links near the top of a page for most items in our system. **NOTE** You will not see one or both of these buttons if you lack permission to perform these operations, which may be due to the status of the item and/or your role in our system.

#. When you click **Create** the first thing you will be asked is to create an alias for your item. This is a lab specific unique identifier for this object taking the form of xxxx:xxxxx where the portion before the colon is a lab designation eg. 4dndcic and the portion after is an identifier that you choose that is unique within your lab group (see section on using aliases here). #.

When you submit your alias you will be brought to a page where you can start entering metadata.

- You will see two gray bars *Fields* and *Linked Objects* and selecting the + will expand those bars to show the fields and objects that can be entered.
- Hovering your pointer over the **i** next to Navigation pops up an explanation for what the different colors of the objects displayed in the Navigation tree.
- If a field or object are required that is indicated.
- The *Fields* section is where you fill out basic fields that are not linked to other database objects.
- In the *Linked Objects* section you can link other Objects to the one you are working on, either by selecting from a list of available existing objects or by creating a new object of the type needed for the particular field it is linked to.
- As you create or add linked Objects you will see the Objects listed in the *Navigate* section change colors accordingly.
- You can use the *Navigate* section to review what you have submitted, validated and what remains to be added.

1. And finally when all your linked objects are submitted and validated (green) you can validate and submit the object to complete your submission.

WARNING: Be careful with the BACK and RELOAD buttons. Currently, if you choose to create a new linked object and then decide you actually don't want to or should have actually chosen an existing object you still should create the object with only the minimum information required, Validate and then Submit it. You will then be taken back to the previous form you were working on and be able to *remove* the unwanted object. If you try to navigate back to the previous page using your browser buttons you will lose the previously unsubmitted changes. We are working to improve this aspect of the interface.

5.2.24.2 Editing Existing Objects

- You can use the online submission interface to make edits to existing items providing you have permission to do so.
- If an object has been 'released' either to the 4DN project or to the public it can no longer be changed.
- If the object has an 'in review' status then you can make changes to fields provided you are the submitter of that object or a submitter for the lab that submitted the object.

WARNING: Please take care to be sure that the object you are editing is really the one you want to change.

1. Navigate to that objects page and if the object is editable then you should see an **Edit** button.

2. After clicking the **Edit** button you will be brought to a page as described above.
3. This time if you click on the **+** in the *Fields* or *Linked Objects* sections you will see the existing values, to which you can make changes as needed.
4. Then validate and submit to commit the changes to the system.