# Spfy Documentation

*Release 4.2.2*

kevinkle

Sep 14, 2018

# Contents:

CHAPTER 1

# Introduction

**Spfy**: Platform for predicting subtypes from E.coli whole genome sequences, and builds graph data for population-wide comparative analyses.

Published as: Le,K.K., Whiteside,M.D., Hopkins,J.E., Gannon,V.P.J., Laing,C.R. Spfy: an integrated graph database for real-time prediction of bacterial phenotypes and downstream comparative analyses. Database (2018) Vol. 2018: article ID bay086; doi:10.1093/database/bay086

Live: https://lfz.corefacility.ca/superphy/spfy/

screenshots/screen-results_list.png

## 1.1 Use:

1. Install Docker (& Docker-Compose separately if you're on Linux, link). mac/windows users have Compose bundled with Docker Engine.
2. `git clone --recursive https://github.com/superphy/spfy.git`
3. `cd spfy/`
4. `docker-compose up`
5. Visit http://localhost:8090
6. Eat cake :cake:

## 1.2 Submodule Build Statuses:

ECTyper: PanPredic: Docker Image for Conda:

## 1.3 Stats:

Comparing different population groups:

screenshots/fishers_overall.png

Runtimes of subtyping modules:

screenshots/spfy_indivs.png

## 1.4 CLI: Generate Graph Files:

- If you wish to only create rdf graphs (serialized as turtle files):

1. First install miniconda and activate the environment from https://raw.githubusercontent.com/superphy/docker-flask-conda/master/app/environment.yml
2. cd into the app folder (where RQ workers typically run from): `cd app/`
3. Run savvy.py like so: `python -m modules/savvy -i tests/ecoli/GCA_001894495.1_ASM189449v1_genomic.fna` where the argument after the `-i` is your genome (FASTA) file.

## 1.5 CLI: Generate Ontology:

screenshots/ontology.png

The ontology for Spfy is available at: https://raw.githubusercontent.com/superphy/backend/master/app/scripts/spfy_ontology.ttl It was generated using https://raw.githubusercontent.com/superphy/backend/master/app/scripts/generate_ ontology.py with shared functions from Spfy's backend code. If you wish to run it, do: 1. `cd app/` 2. `python -m scripts/generate_ontology` which will put the ontology in `app/`

You can generate a pretty diagram from the .ttl file using http://www.visualdataweb.de/webvowl/

## 1.6 CLI: Enqueue Subtyping Tasks w/o Reactapp:

**Note:** currently setup for just .fna files

You can bypass the front-end website and still enqueue subtyping jobs by:

1. First, mount the host directory with all your genome files to `/datastore` in the containers.

   For example, if you keep your files at `/home/bob/ecoli-genomes/`, you'd edit the `docker-compose.yml` file and replace:

   ```
   volumes:
   - /datastore
   ```

   with:

   ```
   volumes:
   - /home/bob/ecoli-genomes:/datastore
   ```

2. Then take down your docker composition (if it's up) and restart it

   ```
   docker-compose down
   docker-compose up -d
   ```

3. Drop and shell into your webserver container (though the worker containers would work too) and run the script.

   ```
   docker exec -it backend_webserver_1 sh
   python -m scripts/sideload
   exit
   ```

Note that reisdues may be created in your genome folder.

## 1.7 Architecture:

| Dock er Imag e | Port s | Name s | Des crip tion |
|---|---|---|---|
| back end- rq | 80/t cp, 443/ tcp | back end_wor ker_1 | the main redi s queu e work ers |
| back end- rq-b laze grap h | 80/t cp, 443/ tcp | back end_wor ker-blaz egra ph-i ds_ 1 | this hand les spfy ID gene rati on for the blaz egra ph data base |
| back end | 0.0. 0.0: 8000 ->80 /tcp , 443/ tcp | back end_web -ngi nx-u wsgi _1 | the flas k back end whic h hand les enqu euei ng task s |
| supe rphy /bla zegr aph: 2.1. 4-in fere ncin g | 0.0. 0.0: 8080 ->80 80/t cp | back end_bla zegr aph_1 | Blaz egra ph Data base |
| redi s:3. 2 | 6379 /tcp | back end_red is_ 1 | Redi s Data base |
| reac tapp | 0.0. 0.0: 8090 ->50 00/t cp | back end_rea ctap p_1 | fron t-en d to spfy |

## 1.8 Further Details:

The `superphy/backend-rq:2.0.0` image is *scalable*: you can create as many instances as you need/have processing power for. The image is responsible for listening to the `multiples` queue (12 workers) which handles most of the tasks, including `RGI` calls. It also listens to the `singles` queue (1 worker) which runs `ECTyper`. This is done as `RGI` is the slowest part of the equation. Worker management in handled in `supervisor`.

The `superphy/backend-rq-blazegraph:2.0.0` image is not scalable: it is responsible for querying the Blazegraph database for duplicate entries and for assigning spfyIDs in *sequential* order. It's functions are kept as minimal as possible to improve performance (as ID generation is the one bottleneck in otherwise parallel pipelines); comparisons are done by sha1 hashes of the submitted files and non-duplicates have their IDs reserved by linking the generated spfyID to the file hash. Worker management in handled in `supervisor`.

The `superphy/backend:2.0.0` which runs the Flask endpoints uses `supervisor` to manage inner processes: `nginx`, `uWsgi`.

## 1.9 Blazegraph:

- We are currently running Blazegraph version 2.1.4. If you want to run Blazegraph separately, please use the same version otherwise there may be problems in endpoint urls / returns (namely version 2.1.1). See #63 Alternatively, modify the endpoint accordingly under `database['blazegraph_url']` in `/app/config.py`

## 1.10 Contributing:

Steps required to add new modules are documented in the Developer Guide.

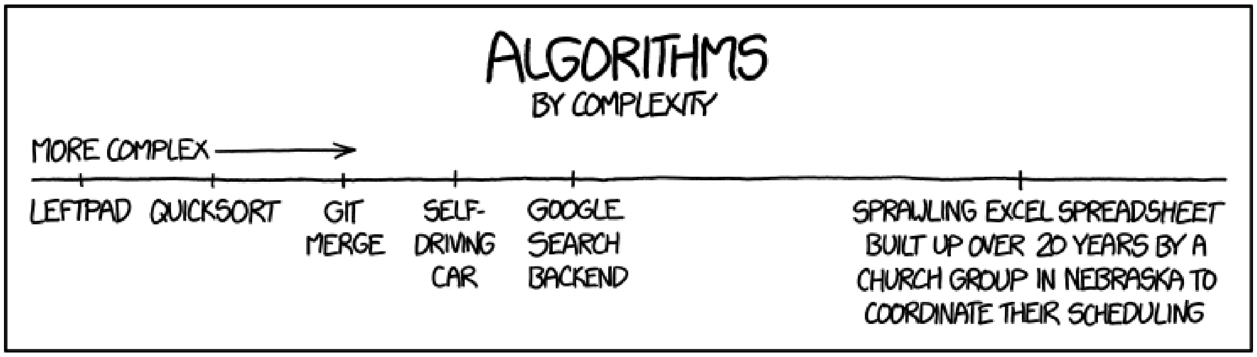
CHAPTER 2

# Developer Guide

**Table of Contents**

## 2.1 Getting Started

Don't worry, genome files are just like Excel spreadsheets.


ALGORITHMS
BY COMPLEXITY
MORE COMPLEX
LEFTPAD
QUICKSORT
GIT MERGE
SELF-DRIVING CAR
GOOGLE SEARCH BACKEND
SPRAWLING EXCEL SPREADSHEET BUILT UP OVER 20 YEARS BY A CHURCH GROUP IN NEBRASKA TO COORDINATE THEIR SCHEDULING

(from the excellent https://xkcd.com/)

We use Docker and Docker-Compose for managing the databases: Blazegraph and Redis, the webserver: Nginx/Flask/Conda, and Redis-Queue (RQ) workers: mostly in Conda. The official Install Docker Compose guide lists steps for installing both the base Docker Engine, and for installing Docker-Compose separately if you're on Linux. For Mac and Windows users, Docker-Compose comes bundled with Docker Engine.

You'll probably also want to install Miniconda as we bundle most dependencies in Conda environments. Specific instructions to Spfy are available at *Installing Miniconda*.

Note that there is a *Debugging* section for tracking down the source of problems you may encounter.

### 2.1.1 Terminology

| Used Interchangeably | Notes |
|---|---|
| jobs, tasks | A job in RQ is typically called a task when discussing the front-end. |
| endpoint, api | We prefer to use endpoint for a route in Flask which the front-end interacts with. |
| spfy, this repo | The superphy/backend repo. |

### 2.1.2 Reading

For the libraries you're not familiar with, we recommend you skim the docs below before starting:

- An overview of HTTP requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
- Flask Blueprints (for routes): http://exploreflask.com/en/latest/blueprints.html
- Redis Queue docs: http://python-rq.org/docs/
- Thinking In React: https://facebook.github.io/react/docs/thinking-in-react.html
- JSX In Depth: https://facebook.github.io/react/docs/jsx-in-depth.html

### 2.1.3 Reference Docs

Javascript:

- Yarn commands for npm users: https://yarnpkg.com/lang/en/docs/migrating-from-npm/
- React Material Design docs: https://react-md.mlaursen.com/components/text-fields
- React-Bootstrap-Table docs: https://allenfang.github.io/react-bootstrap-table/example.html#basic

### 2.1.4 Installing Miniconda

For Linux-64 based distros, grab the Pyhon 2.7 Miniconda install script and install it (be sure to select the option to add Miniconda's path for your .bashrc):

```
wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh
bash Miniconda2-latest-Linux-x86_64.sh
```

Then get, install, and activate our Conda environment.yml:

```
wget https://raw.githubusercontent.com/superphy/docker-flask-conda/master/app/
↪environment.yml
conda env create -f environment.yml
source activate backend
```

### 2.1.5 Genome Files for testing

For testing purposes, we use E.coli genome files from GenBank. A list of ftp links is available at the old github/semantic repo. There should be 5353 genome files in total a .zip of which is available within the NML.

### 2.1.6 Generating More Genomes for Testing

The main points to keep in mind is that Spfy runs quality control checks to ensure submissions are E.coli and that hash checking is employed to avoid duplicate entries in the datbase. The way we generate fakes are using a seed folder of actual genomes (to pass QC) and renmaing the contig headers (to pass hash checking).

Usage:

1. Activate the conda env described in *Installing Miniconda*.
2. cd in `backend/scripts/` (not: ``backend/app/scripts`)
3. Run: `python generate_fakegenomes.py -i ~/ecoli-genomes/ -n 50000` where `-i` gives the seed folder and `-n` gives the number of genomes to generate. This will put all the fakes in `~/ecoli-genomes/fakes/`.

### 2.1.7 Docker Caveats

We've had problems in the past with Ubuntu Desktop versions 16.04.2 LTS and 17.04, and Ubuntu Server 16.04.2 LTS not connecting to NPM when building Docker images and from within the building. Builds work fine with Ubuntu Server 16.04.2 LTS on Cybera and for Ubuntu Server 12.04 and 14.04 LTS on Travis-CI. Within the building, RHEL-based operating systems (CentOS / Scientific Linux) build our NPM-dependent images (namely, reactapp) just fine. Tested the build at home on Ubuntu Server 16.04.2 LTS and it works fine - looks like this is isolated to within the buildng @NML Lethbridge.

**Warning:** As of June 30, 2017 Ubuntu Server 16.04.2 LTS is building NPM-dependent images okay @NML Lethbridge.

**Note:** In general, we recommend you run Docker on Ubuntu 16.04.2 LTS (Server or Desktop) if you're outside the NML's Lethrbidge location. Otherwise, CentOS is a secondary option.

For RHEL-based OSs, I don't recommend using *devicemapper*, but instead use *overlayfs*. Reasons are documented at https://github.com/moby/moby/issues/3182. There is a guide on setting up Docker with *overlayfs* at https://dcos.io/docs/1.7/administration/installing/custom/system-requirements/install-docker-centos/, though I haven't personally tested it. UPDATE: (June 22'17) There is a guide written by a Red Hat dev. http://www.projectatomic.io/blog/2015/06/notes-on-fedora-centos-and-docker-storage-drivers/

If you do end up using *devicemapper* and run into disk space issues, such as:

```
172.18.0.1 - - [05/Jun/2017:17:50:01 +0000] "GET / HTTP/1.1" 200 12685 "-" "Mozilla/5.
↪0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110␣
↪Safari/537.36" "-"
2017/06/05 17:50:13 [warn] 11#11: *2 a client request body is buffered to a temporary␣
↪file /var/cache/nginx/client_temp/0000000001, client: 172.18.0.1, server: ,␣
↪request: "POST /upload HTTP/1.1", host: "localhost:8000", referrer: "http://
↪localhost:8000/"
[2017-06-05 17:58:31,417] ERROR in app: Exception on /upload [POST]
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask/app.py", line 1982,␣
↪in wsgi_app
    response = self.full_dispatch_request()
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask/app.py", line 1614,␣
↪in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask_cors/extension.py",␣
↪line 161, in wrapped_function
    return cors_after_request(app.make_response(f(*args, **kwargs)))
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask/app.py", line 1517,␣
↪in handle_user_exception
    reraise(exc_type, exc_value, tb)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask/app.py", line 1612,␣
↪in full_dispatch_request
    rv = self.dispatch_request()
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask/app.py", line 1598,␣
↪in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
  File "./routes/views.py", line 86, in upload
    form = request.form
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/local.py", line␣
↪343, in __getattr__
    return getattr(self._get_current_object(), name)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/utils.py", line␣
↪73, in __get__
    value = self.func(obj)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/wrappers.py",␣
↪line 492, in form
    self._load_form_data()
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/flask/wrappers.py", line␣
↪185, in _load_form_data
    RequestBase._load_form_data(self)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/wrappers.py",␣
↪line 361, in _load_form_data
    mimetype, content_length, options)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/formparser.py",␣
↪line 195, in parse
    content_length, options)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/formparser.py",␣
↪line 100, in wrapper
    return f(self, stream, *args, **kwargs)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/formparser.py",␣
↪line 212, in _parse_multipart
    form, files = parser.parse(stream, boundary, content_length)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/formparser.py",␣
↪line 523, in parse
    return self.cls(form), self.cls(files)
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/datastructures.py
↪", line 384, in __init__
    for key, value in mapping or ():
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/formparser.py",␣
↪line 521, in <genexpr>
    form = (p[1] for p in formstream if p[0] == 'form')
  File "/opt/conda/envs/backend/lib/python2.7/site-packages/werkzeug/formparser.py",␣
↪line 497, in parse_parts
    _write(ell)
```

(continues on next page)

(continued from previous page)

```
IOError: [Errno 28] No space left on device
[pid: 44|app: 0|req: 2/2] 172.18.0.1 () {46 vars in 867 bytes} [Mon Jun  5 17:53:08␣
↪2017] POST /upload => generated 291 bytes in 323526 msecs (HTTP/1.1 500) 2 headers␣
↪in 84 bytes (54065 switches on core 0)
172.18.0.1 - - [05/Jun/2017:17:58:32 +0000] "POST /upload HTTP/1.1" 500 291 "http://
↪localhost:8000/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like␣
↪Gecko) Chrome/58.0.3029.110 Safari/537.36" "-"
```

Which was displayed by running:

```
docker-compose logs backend_webserver_1
```

You will have to increase the volume disk sizes: https://forums.docker.com/t/increase-container-volume-disk-size/1652/8

```
# With Centos 7 I did the following to increase the default size of the containers
# Modify the docker config in /etc/sysconfig/docker-storage to add the line:
DOCKER_STORAGE_OPTIONS= - -storage-opt dm.basesize=20G
service docker stop
rm /var/lib/docker NOTE THIS DELETES ALL IMAGES etc. SO MAKE A BACKUP
service docker start
docker load < [each_save_in_backup.tar]
docker run -i -t [imagename] /bin/bash
# In the bash prompt of the docker container "df -k" should show 20GB / file system␣
↪size now.
```

### 2.1.8 Redis

**Warning:** By default, our docker composition is setup to run Redis db with persistant storage so jobs are kept even in you stop and restart the `redis` service. This is useful in production and regular usage scenarios as all your jobs are not lost if the composition is stopped or the server/computer is rebooted. However, this also means that if you write a job which errors out and also upload a bunch of files, they will continue to be started even if you stop the composition to write fixes.

To run Redis in non-persistant mode, in `docker-compose.yml` replace:

```
redis:
  image: redis:3.2
  command: redis-server --appendonly yes # for persistance
  volumes:
  - /data
```

with:

```
redis:
  image: redis:3.2
```

### 2.1.9 The General Workflow

**Note:** To use `docker-compose` commands, you must be in the same directory as the `docker-compose.yml` file you're trying to work with. This is because Docker-Compose uses that .yml file to determine the names of services you're running commands against; for example you might run `docker-compose logs webserver`. You can still access the underlying docker containers outside of the folder by interfacing with the docker engine directly: `docker logs backend_webserver_1`.

For working on the backend:

1. Make your changes/additions
2. Rebuild the images

   ```
   docker-compose build --no-cache
   ```

   or selectively:

   ```
   docker-compose build --no-cache webserver worker
   ```

3. Bring up the composition and use Chrome's devtools for testing

   ```
   docker-compose up
   ```

4. Check logs as appropriate:

   ```
   docker-compose logs webserver
   docker-compose logs worker
   ```

5. Cleanup the composition you just started

   ```
   docker-compose down
   ```

6. Make more changes and rebuild

   ```
   docker-compose build --no-cache
   ```

For working on the front-end:

We reccomend using `yarn start` as it has hot-reloading enabled so it'll automatically rebuild and display your changes at `localhost:3000`.

1. First, start up the backend (if you're now making changes to the backend, we'll use the default build step when bringing up the composition)

   ```
   docker-compose up
   ```

2. In a separate terminal, fork and clone the reactapp repo, and then bring it up (you'll have to install `node` and `yarn`:

   ```
   yarn install
   yarn start
   ```

3. Make changes to your fork of reactapp and you'll see them refreshed live at `localhost:3000`.

## 2.2 Adding a New Module

There are a few ways of adding a new module:

1. Integrate your code into the Spfy codebase and update the RQ workers accordingly.
2. Add a enqueuing method to Spfy's code, but then create a new queue and a new docker image, with additional dependencies, which is added to Spfy's docker-compose.yml file.
3. Setting up your module as a microservice running in its own Docker container, add a worker to handle requests to RQ.

**Note:** The quickest approach is to integrate your code into the Spfy codebase and update the RQ workers accordingly.

If you wish to integrate your code with Spfy, you'll have to update any dependencies to the underlying Conda-based image the RQ workers depend on. You'll also have to include your code in the */app* directory of this repo, as that is the only directory the current RQ workers contain. The intended structure is to create a directory in */app/modules* for your codebase and a *.py* file above at */app/modules/newmodule.py*, for example, which contains the method your *Queue.enqueue()* function uses.

There is more specific documentation for this process in *Directly Adding a New Module*.

If you wish to create your own image, you can use the RQ worker image as a starting point. Specifically you'll want to add your repo as a git submodule in *superphy/backend* and modify the *COPY ./app /app* to target your repo, similar to the way reactapp is included. You'll also want to take a look at the supervisord-rq.conf which controls the RQ workers.

In both cases, the spfy webserver will have to be modified in order for the front-end to have an endpoint target; this is documented in *Adding an Endpoint in Flask*. The front-end will also have to be modified for there to be a form to submit tasks and have a results view generated for your new module; this is documented in *Modifying the Front-End*.

## 2.3 Directly Adding a New Module

**Warning:** Everything (rq workers, uwsgi, etc.) run inside `/app`, and all python imports should be relative to this. Such as

```
from middleware.blazegraph.reserve_id import reserve_id
```

The top-most directory is used to build Docker Images and copies the contents of `/app` to run inside the containers. This is done as the apps (Flask, Reactapp) themselves don't need copies of the Dockerfiles, other apps, etc.

### 2.3.1 About the Existing Codebase

If you want to store the results to Blazegraph, you can add that to your pipeline. For subtyping tasks (ECTyper, RGI), the graph generation is handled in `/app/modules/turtleGrapher/datastruct_savvy.py`, you can use that as an example. Note that the `upload_graph()` call is made within `datastruct_savvy.py`; this is done to avoid having to pass the resulting `rdflib.Graph` object between tasks. Also, the base graph (only containing information about the file, without any results from analyses) is handled by `/app/modules/turtleGrapher/turtle_grapher.py`.

### 2.3.2 Adding Dependencies via Conda

The main environment.yml file is located in our superphy/docker-flask-conda repo. This is used by the worker and worker-blazegraph-ids containers (and the webserver container, though that may/should change). We also pull this base superphy/docker-flask-conda image from Docker Hub. So you would have to:

1. push the new image
2. specify the new version on each Dockerfile, namely via the

```
FROM superphy/docker-flask-conda:2.0.0
```

tag.

To get started, install Miniconda and clone the docker-flask-conda repo:

```
git clone https://github.com/superphy/docker-flask-conda.git && cd docker-flask-conda
```

Recreate the env:

```
conda env create -f app/environment.yml
```

Activate the env:

```
source activate backend
```

Then you can install any dependencies as usual. Via pip:

```
pip install whateverpackage
```

or conda

```
conda install whateverpackage
```

You can then export the env:

```
conda env export > app/environment.yml
```

If you push your changes to github on *master*, Travis-CI is setup to build the Docker Image and push it to Docker Hub automatically under the tag *latest*.

Otherwise, build and push the image under your own tag, for example *0.0.1*:

```
docker build -t superphy/docker-flask-conda:0.0.1 .
docker push superphy/docker-flask-conda:0.0.1
```

Then specific your image in the corresponding Dockerfiles: worker. If you're adding dependencies to flask, also update the webserver Dockerfile.

```
FROM superphy/docker-flask-conda:0.0.1
```

### 2.3.3 Integrating your Codebase into Spfy

There are two ways of approaching this:

1. If you're not using any of Spfy's codebase, add your code as a git submodule in */app/modules/*
2. If you are using Spfy's codebase, fork and create a directory in */app/modules/* with your code.

In both cases, you should add a method in */app/module/pickaname.py* which enqueues a call to your package. More information on this is documented at *Enqueing a Job to RQ*.

To add a git submodule, clone the repo and create a branch:

```
git clone --recursive https://github.com/superphy/backend.git && cd backend/
git checkout -b somenewmodule
```

You can then add your repo and commit it to *superphy/backend* as usual:

```
git submodule add https://github.com/chaconinc/DbConnector app/modules/DbConnector
git add .
git commit -m 'ADD: my new module'
```

or a specific branch:

```
git submodule add -b somebranch https://github.com/chaconinc/DbConnector app/modules/
↪DbConnector
```

Note that the main repo *superphy/backend* will pin your git submodule to a specific commit. You can update it to the HEAD of w/e branch was used by running a *git pull* from within the submodule's directory and then adding it in the main repo. If you push this change to GitHub, to update other clones of superphy/backend run:

```
git submodule update
```

## 2.4 Adding an Endpoint in Flask

To create a new endpoint in Flask, you'll have to:

1. Create a Blueprint with your route(s) and register it to the app.
2. Enqueue a job in RQ
3. Return the job id via Flask to the front-end

We recommend you perform the setup in *Monitoring RQ* before you begin.

### 2.4.1 Creating a Blueprint

We use Flask Blueprints to compartmentalize all routes. They are contained in */app/routes* and have the following basic structure:

```
from flask import Blueprint, request, jsonify

bp_someroutes = Blueprint('someroutes', __name__)

# if methods is not defined, default only allows GET
@bp_someroutes.route('/api/v0/someroute', methods=['POST'])
def someroute():
  form = request.form
  return jsonify('Got your form')
```

Note that a blueprint can have multiple routes defined in it such as in ra_views.py which is used to build the group options for Fisher's comparison. To add a new route, create a python file such as */app/routes/someroutes.py* with the above structure. Then in the app factory.py import your blueprint via:

```
from routes.someroute import bp_someroute
```

and register your blueprint in *create_app()* by adding:

```
app.register_blueprint(bp_someroute)
```

Note that we allow CORS on all routes of form */api/** such as */api/v0/someroute*. This is required as the front-end reactapp is deployed in a separate container (and has a separate IP Address) from the Flask app.

### 2.4.2 Enqueing a Job to RQ

You will then have to enqueue a job, based off that request form. There is an example of how form parsing is handled for Subtyping in the *upload()* method of ra_posts.py.

If you're integrating your codebase with Spfy, add your code to a new directory in */app/modules* and a method which handles enqueing in */app/modules/somemodule.py* for example. The gc.py file resembles a basic template for a method to enqueue.

```
import logging
import config
import redis
from rq import Queue
from modules.comparisons.groupcomparisons import groupcomparisons
from modules.loggingFunctions import initialize_logging

# logging
log_file = initialize_logging()
log = logging.getLogger(__name__)

redis_url = config.REDIS_URL
redis_conn = redis.from_url(redis_url)
multiples_q = Queue('multiples', connection=redis_conn, default_timeout=600)

def blob_gc_enqueue(query, target):
    job_gc = multiples_q.enqueue(groupcomparisons, query, target, result_ttl=-1)
    log.info('JOB ID IS: ' + job_gc.get_id())
    return job_gc.get_id()
```

Of note is that when calling RQ's enqueue() method, a custom Job class is returned. It is important that our enqueuing method returns the job id to flask, which is typically some hash such as:

```
16515ba5-040d-4315-9c88-a3bf5bfbe84e
```

### 2.4.3 Returning the Job ID to the Front-End

Generally, we expect the return from Flask (to the front-end) to be a dictionary with the job id as the key to another dictionary with keys *analysis* and *file* (if relevant), though this is not strictly required (a single line containing the key will also work, as you handle naming of analysis again when doing a *dispatch()* in reactapp - more on this later). For example, a return might be:

```
"c96619b8-b089-4a3a-8dd2-b09b5d5e38e9": {
  "analysis": "Virulence Factors and Serotype",
  "file": "/datastore/2017-06-14-21-26-43-375215-GCA_001683595.1_NGF2_genomic.fna"
}
```

It is expected that only 1 job id be returned per request. In v4.2.2 we introduced the concept of *blob* ids in which dependency checking is handled server-side; you can find more details about this in reactapp issue #30 and backend issue #90. The Redis DB was also set to run in persistent-mode, with results stored to disk inside a docker volume. The *blob* concept is only relevant if you handle parallelism & pipelines for a given task (ex. Subtyping) through multiple RQ jobs (ex. QC, ID Reservation, ECTyper, RGI, parsing, etc.); if you handle parallelism in your own codebase, then this isn't required.

Another point to note is that the:

```
result_ttl=-1
```

parameter in the *enqueue()* method is required to store the result in Redis permanently; this is done so results will forever be available to the front-end. If we ever scale Spfy to widespread usage, it may be worth setting a ttl of 48 hours or so via:

```
result_ttl=172800
```

where the ttl is measured in seconds. A warning message would also have to be added to reactapp.

### 2.4.4 Seeing Your Changes in Docker

To rebuild the Flask image, in */backend*:

```
docker-compose stop webserver worker
docker-compose build --no-cache webserver worker
docker-compose up
```

## 2.5 Optional: Adding a new Queue

Normally, we distribute tasks between two main queues: *singles* and *multiples*. The singles queue is intended for tasks that can't be run in parallel within the same container (though you can probably run multiple containers, if you so wish); our use-case is for ECTyper. Everything else is intended to be run on the *multiples* queue.

If you wish to add your own Queue, you'll have to create some worker to listen to it. Ideally, do this by creating a new Docker container for your worker by copying the worker Dockerfile as your starting point then copying and modifying the supervisord-rq.conf to listen to your new queue. Specifically, the:

```
command=/opt/conda/envs/backend/bin/rq worker -c config multiples
```

would have to be modified to target the name of the new Queue your container listens to; by replacing *multiples* with *newqueue*, for example.

Eventually, we may wish to add priority queues once the number of tasks become large and we have long-running tasks alongside ones that should immediately return to the user. This can be defined by the order in which queues are named in the supervisord command:

```
command=/opt/conda/envs/backend/bin/rq worker -c config multiples
```

For example, queues *dog* and *cat* can be ordered:

```
command=/opt/conda/envs/backend/bin/rq worker -c config dog cat
```

which instructs the RQ workers to run tasks in *dog* first, before running tasks in *cat*.

## 2.6 Modifying the Front-End

I'd recommend you leave Spfy's setup running in Docker-Compose and run the reactapp live so you can see immediate updates.

To get started, install node and then install yarn. For debugging, I also recommend using Google Chrome and installing the React Dev Tools and Redux Dev Tools.

> Optionally, I like to run Spfy's composition on one of the Desktops while coding away on my laptop. You can do the same by modifying *ROOT* api address in api.js to point to a different IP address or name:
>
> ```
> const ROOT = 'http://10.139.14.212:8000/'
> ```

Then, with Spfy's composition running, you'll want to clone reactapp and run:

```
cd reactapp/
yarn install
yarn start
```

Our reactapp uses *Redux* to store jobs, but also uses regular *React states* when building forms or displaying results. This was done so you don't have to be too familiar with Redux when building new modules. The codebase is largely JSX+ES6.

### 2.6.1 Adding a New Task Card

The first thing you'll want to do is add a description of your module to api.js. For example, the old analyses const is:

```
export const analyses = [{
  'analysis':'subtyping',
  'description':'Serotype, Virulence Factors, Antimicrobial Resistance',
  'text':(
    <p>
      Upload genome files & determine associated subtypes.
      <br></br>
      Subtyping is powered by <a href="https://github.com/phac-nml/ecoli_serotyping">
↪ECTyper</a>.
      AMR is powered by <a href="https://card.mcmaster.ca/analyze/rgi">CARD</a>.
    </p>
  )
},{
  'analysis':'fishers',
  'description':"Group comparisons using Fisher's Exact Test",
  'text':'Select groups from uploaded genomes & compare for a chosen target datum.'
}]
```

If we added a new module called *ml*, analyses might be:

```
export const analyses = [{
  'analysis':'subtyping',
  'description':'Serotype, Virulence Factors, Antimicrobial Resistance',
  'text':(
    <p>
      Upload genome files & determine associated subtypes.
      <br></br>
      Subtyping is powered by <a href="https://github.com/phac-nml/ecoli_serotyping">
↪ECTyper</a>.
```

(continues on next page)

(continued from previous page)

```
      AMR is powered by <a href="https://card.mcmaster.ca/analyze/rgi">CARD</a>.
    </p>
  )
},{
  'analysis':'fishers',
  'description':"Group comparisons using Fisher's Exact Test",
  'text':'Select groups from uploaded genomes & compare for a chosen target datum.'
},{
  'analysis':'ml',
  'description': "Machine learning module for Spfy",
  'text': 'Multiple machine learning algorithms such as, support vector machines,␣
↪naive Bayes, and the Perceptron algorithm.'
}]
```

This will create a new card for in tasks at the root page.

### 2.6.2 Adding a New Task Form

**Note:** On terminology: we consider *containers* to be *Redux-aware*; that is, they require the *connect()* function from *react-redux*. *Components* are generally not directly connected to Redux and instead get information from the Redux store passed down to it via the component's *props*. Note that this is not strictly true as we make use of *react-refetch*, which is a fork of Redux and uses a separate *connect()* function, to poll for job statuses and results. However, the interaction between *react-refetch* and *redux* is largely abstracted away from you and instead maps a components props directly to updates via *react-refetch* - you don't have to dispatch actions or pull down updates separately.

Then create a container in */src/containers* which will be your request form. You can look at Subtyping.js for an example.

```
import React, { PureComponent } from 'react';
// react-md
import FileInput from 'react-md/lib/FileInputs';
import Checkbox from 'react-md/lib/SelectionControls/Checkbox'
import TextField from 'react-md/lib/TextFields';
import Button from 'react-md/lib/Buttons';
import Switch from 'react-md/lib/SelectionControls/Switch';
import Subheader from 'react-md/lib/Subheaders';
import CircularProgress from 'react-md/lib/Progress/CircularProgress';
// redux
import { connect } from 'react-redux'
import { addJob } from '../actions'
import { subtypingDescription } from '../middleware/subtyping'
// axios
import axios from 'axios'
import { API_ROOT } from '../middleware/api'
// router
import { Redirect } from 'react-router'
import Loading from '../components/Loading'

class Subtyping extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      file: null,
```

(continues on next page)

(continued from previous page)

```
      pi: 90,
      amr: false,
      serotype: true,
      vf: true,
      submitted: false,
      open: false,
      msg: '',
      jobId: "",
      hasResult: false,
      groupresults: true,
      progress: 0
    }
  }
  _selectFile = (file) => {
    console.log(file)
    if (!file) { return; }
    this.setState({ file });
  }
  _updatePi = (value) => {
    this.setState({ pi: value });
  }
  _updateSerotype = (value) => {
    this.setState({ serotype: value })
  }
  _updateAmr = (value) => {
    this.setState({ amr: value })
  }
  _updateVf = (value) => {
    this.setState({ vf: value })
  }
  _updateGroupResults = (groupresults) => {
    this.setState({ groupresults })
  }
  _updateUploadProgress = ( progress ) => {
    this.setState({progress})
  }
  _handleSubmit = (e) => {
    e.preventDefault() // disable default HTML form behavior
    // open and msg are for Snackbar
    // uploading is to notify users
    this.setState({
      uploading: true
    });
    // configure a progress for axios
    const createConfig = (_updateUploadProgress) => {
      var config = {
        onUploadProgress: function(progressEvent) {
          var percentCompleted = Math.round( (progressEvent.loaded * 100) /␣
↪progressEvent.total );
          _updateUploadProgress(percentCompleted)
        }
      }
      return config
    }
    // create form data with files
    var data = new FormData()
    // eslint-disable-next-line
```

(continues on next page)

(continued from previous page)

```
    this.state.file.map((f) => {
      data.append('file', f)
    })
    // append options
    // to match spfy(angular)'s format, we dont use a dict
    data.append('options.pi', this.state.pi)
    data.append('options.amr', this.state.amr)
    data.append('options.serotype', this.state.serotype)
    data.append('options.vf', this.state.vf)
    // new option added in 4.2.0, group all files into a single result
    // this means polling in handled server-side
    data.append('options.groupresults', this.state.groupresults)
    // put
    axios.post(API_ROOT + 'upload', data, createConfig(this._updateUploadProgress))
      .then(response => {
        console.log(response)
        // no longer uploading
        this.setState({
          uploading: false
        })
        let jobs = response.data
        // handle the return
        for(let job in jobs){
          let f = (this.state.file.length > 1 ?
          String(this.state.file.length + ' Files')
          :this.state.file[0].name)
          if(jobs[job].analysis === "Antimicrobial Resistance"){
            this.props.dispatch(addJob(job,
              "Antimicrobial Resistance",
              new Date().toLocaleTimeString(),
              subtypingDescription(f, this.state.pi, false, false, this.state.amr)
            ))
          } else if (jobs[job].analysis === "Virulence Factors and Serotype") {
            let descrip = ''
            if (this.state.vf && this.state.serotype){descrip = "Virulence Factors␣
↪and Serotype"}
            else if (this.state.vf && !this.state.serotype) {descrip = "Virulence␣
↪Factors"}
            else if (!this.state.vf && this.state.serotype) {descrip = "Serotype"}
            this.props.dispatch(addJob(job,
              descrip,
              new Date().toLocaleTimeString(),
              subtypingDescription(f, this.state.pi, this.state.serotype, this.state.
↪vf, false)
            ))
          } else if (jobs[job].analysis === "Subtyping") {
            // set the jobId state so we can use Loading
            const jobId = job
            this.setState({jobId})
            // dispatch
            this.props.dispatch(addJob(job,
              "Subtyping",
              new Date().toLocaleTimeString(),
              subtypingDescription(
                f , this.state.pi, this.state.serotype, this.state.vf, this.state.amr)
            ))
          }
```

(continues on next page)

(continued from previous page)

```
        }
        const hasResult = true
        this.setState({hasResult})
      })
  };
  render(){
    const { file, pi, amr, serotype, vf, groupresults, uploading, hasResult, progress␣
↪} = this.state
    return (
      <div>
        {/* uploading bar */}
        {(uploading && !hasResult) ?
          <div>
            <CircularProgress key="progress" id="loading" value={progress} centered=
↪{false} />
            Uploading... {progress} %
          </div>
          : ""
        }
        {/* actual form */}
        {(!hasResult && !uploading)?
          <form className="md-text-container md-grid">
            <div className="md-cell md-cell--12">
              <FileInput
                id="inputFile"
                secondary
                label="Select File(s)"
                onChange={this._selectFile}
                multiple
              />
              <Switch
                id="groupResults"
                name="groupResults"
                label="Group files into a single result"
                checked={groupresults}
                onChange={this._updateGroupResults}
              />
              {!groupresults ?
                <Subheader primaryText="(Will split files & subtyping methods into␣
↪separate results)" inset />
              : ''}
              <Checkbox
                id="serotype"
                name="check serotype"
                checked={serotype}
                onChange={this._updateSerotype}
                label="Serotype"
              />
              <Checkbox
                id="vf"
                name="check vf"
                checked={vf}
                onChange={this._updateVf}
                label="Virulence Factors"
              />
              <Checkbox
                id="amr"
```

(continues on next page)

(continued from previous page)

```
                name="check amr"
                checked={amr}
                onChange={this._updateAmr}
                label="Antimicrobial Resistance"
              />
              {amr ?
                <Subheader primaryText="(Note: AMR increases run-time by several␣
↪minutes per file)" inset />
              : ''}
              <TextField
                id="pi"
                value={pi}
                onChange={this._updatePi}
                helpText="Percent Identity for BLAST"
              />
              <Button
                raised
                secondary
                type="submit"
                label="Submit"
                disabled={!file}
                onClick={this._handleSubmit}
              />
            </div>
            <div className="md-cell md-cell--12">
              {this.state.file ? this.state.file.map(f => (
                <TextField
                  key={f.name}
                  defaultValue={f.name}
                />
              )) : ''}
            </div>
          </form> :
          // if results are grouped, display the Loading page
          // else, results are separate and display the JobsList cards page
          (!uploading?(!groupresults?
            <Redirect to='/results' />:
            <Loading jobId={this.state.jobId} />
          ):"")
        }
      </div>
    )
  }
}

Subtyping = connect()(Subtyping)

export default Subtyping
```

The important part to note is the form submission:

```
axios.post(API_ROOT + 'upload', data, createConfig(this._updateUploadProgress))
      .then(response => {
        console.log(response)
        // no longer uploading
        this.setState({
          uploading: false
```

(continues on next page)

(continued from previous page)

```
        })
        let jobs = response.data
        // handle the return
        for(let job in jobs){
          let f = (this.state.file.length > 1 ?
          String(this.state.file.length + ' Files')
          :this.state.file[0].name)
          if(jobs[job].analysis === "Antimicrobial Resistance"){
            this.props.dispatch(addJob(job,
              "Antimicrobial Resistance",
              new Date().toLocaleTimeString(),
              subtypingDescription(f, this.state.pi, false, false, this.state.amr)
            ))
```

(truncated)

We can take a look at a simpler example in Fishers.js where there aren't multiple *jobs[job].analysis* === *"Antimicrobial Resistance"* analysis types in a single form.

```
axios.post(API_ROOT + 'newgroupcomparison', {
      groups: groups,
      target: target
    })
      .then(response => {
        console.log(response);
        const jobId = response.data;
        const hasResult = true;
        this.setState({jobId})
        this.setState({hasResult})
        // add jobid to redux store
        this.props.dispatch(addJob(jobId,
          'fishers',
          new Date().toLocaleTimeString(),
          fishersDescription(groups, target)
        ))
      });
```

First you'd want to change the POST route so it targets your new endpoint.

```
axios.post(API_ROOT + 'someroute', {
```

Note that *API_ROOT* prepends the *api/v0/* so the full route might be *api/v0/someroute*.

Now we need to dispatch an *addJob* action to Redux. This stores the job information in our Redux store, under the *jobs* list. In our example, we used a function to generate the description, but if you were to add a dispatch for your *ml* module you might do something like:

```
axios.post(API_ROOT + 'someroute', {
        groups: groups,
        target: target
      })
        .then(response => {
          console.log(response);
          const jobId = response.data;
          const hasResult = true;
          this.setState({jobId})
          this.setState({hasResult})
```

(continues on next page)

(continued from previous page)

```
          // add jobid to redux store
          this.props.dispatch(addJob(jobId,
            'ml',
            new Date().toLocaleTimeString(),
            'my description of what ml options were chosen'
          ))
        });
```

Then, after creating your form, in /src/containers/App.js add an import for your container:

```
import ML from '../containers/ML'
```

then add a route:

```
<Switch key={location.key}>
   <Route exact path="/" location={location} component={Home} />
   <Route path="/fishers" location={location} component={Fishers} />
   <Route path="/subtyping" location={location} component={Subtyping} />
   <Route exact path="/results" location={location} component={Results} />
   <Route path="/results/:hash" location={location} component={VisibleResult} />
 </Switch>
```

would become:

```
<Switch key={location.key}>
   <Route exact path="/" location={location} component={Home} />
   <Route path="/fishers" location={location} component={Fishers} />
   <Route path="/subtyping" location={location} component={Subtyping} />
   <Route path="/ml" location={location} component={ML} />
   <Route exact path="/results" location={location} component={Results} />
   <Route path="/results/:hash" location={location} component={VisibleResult} />
 </Switch>
```

Now your form will render at */ml*.

## 2.6.3 Adding a Results Page

When your form dispatches an *addJob* action to Redux, the */results* page will automatically populate and poll for the status of your job. You'll now need to add a component to display the results to the user. For tabular results, we use the react-bootstrap-table package. You can look at /src/components/ResultsFishers.js as a starting point.

```
import React, { Component } from 'react';
import { connect } from 'react-refetch'
// progress bar
import CircularProgress from 'react-md/lib/Progress/CircularProgress';
// requests
import { API_ROOT } from '../middleware/api'
// Table
import { BootstrapTable, TableHeaderColumn } from 'react-bootstrap-table';

class ResultFishers extends Component {
  render() {
    const { results } = this.props
    const options = {
      searchPosition: 'left'
```

(continues on next page)

(continued from previous page)

```
    };
    if (results.pending){
      return <div>Waiting for server response...<CircularProgress key="progress" id=
↪'contentLoadingProgress' /></div>
    } else if (results.rejected){
      return <div>Couldn't retrieve job: {this.props.jobId}</div>
    } else if (results.fulfilled){
      console.log(results)
      return (
        <BootstrapTable data={results.value.data} exportCSV search options={options}>
          <TableHeaderColumn  isKey dataField='0' dataSort filter={ { type:
↪'TextFilter', placeholder: 'Please enter a value' } } width='400' csvHeader='Target
↪'>Target</TableHeaderColumn>
          <TableHeaderColumn  dataField='1' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } csvHeader='QueryA'>QueryA</
↪TableHeaderColumn>
          <TableHeaderColumn  dataField='2' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } csvHeader='QueryB'>QueryB</
↪TableHeaderColumn>
          <TableHeaderColumn  dataField='3' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } width='140' csvHeader='#Present QueryA'>
↪#Present QueryA</TableHeaderColumn>
          <TableHeaderColumn  dataField='4' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } width='140' csvHeader='#Absent QueryA'>
↪#Absent QueryA</TableHeaderColumn>
          <TableHeaderColumn  dataField='5' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } width='140' csvHeader='#Present QueryB'>
↪#Present QueryB</TableHeaderColumn>
          <TableHeaderColumn  dataField='6' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } width='140' csvHeader='#Absent QueryB'>
↪#Absent QueryB</TableHeaderColumn>
          <TableHeaderColumn  dataField='7' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } width='140' csvHeader='P-Value'>P-Value</
↪TableHeaderColumn>
          <TableHeaderColumn  dataField='8' dataSort filter={ { type: 'TextFilter',␣
↪placeholder: 'Please enter a value' } } width='140' csvHeader='Odds Ratio'>Odds␣
↪Ratio</TableHeaderColumn>
        </BootstrapTable>
      );
    }
  }
}

export default connect(props => ({
  results: {url: API_ROOT + `results/${props.jobId}`}
}))(ResultFishers)
```

In the case of Fisher's, the response from Flask is generated by the:

```
df.to_json(orient='split')
```

from the Pandas DataFrame. This creates an object with keys *columns*, *data*, and *index*. In particular, under the *data* key is an array of arrays:

```
[["https:\/\/www.github.com\/superphy#hlyC","O111","O24",1.0,0.0,0.0,1.0,null,1.0],[
↪"https:\/\/www.github.com\/superphy#hlyB","O111","O24",1.0,0.0,0.0,1.0,null,1.0],[
↪"https:\/\/www.github.com\/superphy#hlyA","O111","O24",1.0,0.0,0.0,1.0,null,1.0]]
```

(only an example, the full results.value.data array is 387 arrays long, and can vary)

Note that we use

```
dataField='5'
```

for example, which we apply to:

```
csvHeader='#Present QueryB'
```

which is used for exporting to .csv. And in between the TableHeaderColumn tags:

```
<TableHeaderColumn>#Present QueryB</TableHeaderColumn>
```

(options removed)

The *#Present QueryB* is used when displaying the webpage.

Finally, in /src/components/ResultsTemplates.js import you component:

```
import ResultML from './ResultML'
```

and add the case to the switch which decides which result view to return:

```
case "ml":
    return <ML jobId={job.hash} />
```

## 2.7 Packaging It All Together

Once the main *superphy/backend* repo has any submodule you specified at the correct head, you can rebuild the entire composition by running:

```
git submodule update
docker-compose build --no-cache .
docker-compose up
```

Alternatively, to run docker-compose in detached-head mode (where the composition runs entirely by the Docker daemon, without need for a linked shell), run:

```
docker-compose up -d
```

## 2.8 Adding a New Option to the Subtyping Module

While reviewing *Adding a New Module* is important to see the general workflow, if you're modifying the Subtyping task to add a new analysis option you'll have to *modify* the existing codebase instead of simply *adding* a new module. There are a few things you'll have to do:

1. Add a Switch to the Subtyping.js and ensure the selection is appended to the formData
2. Handle the selected option in the `upload()` function in ra_posts.py
3. Create an enqueue() call in spfy.py
4. Create a folder or git submodule in `app/modules` which contains the rest of the code your option needs

5. If you want to return the results to the front-end or upload the results to blazegraph, you'll have to parse your return to fit the format of datastruct_savvy.py and then enqueue the datastruct_savvy() call with your results as the arg and all that job to the `jobs` dict in `upload()` of *ra_posts.py*
6. Then we need to edit beautify.py to parse the same dict used for datastruct_savvy.py. Afterwhich, the `merge_job_results()` in ra_statuses.py will automatically merge the result and return it to the front-end

### 2.8.1 Adding a Checkbox to the Subtyping.js

As shown in Subtyping.js , checkboxes are defined like so:

```
<Checkbox
  id="serotype"
  name="check serotype"
  checked={serotype}
  onChange={this._updateSerotype}
  label="Serotype"
/>
```

The important points are the `checked={serotype}` where `serotype` refers to a state defined by:

```
constructor(props) {
  super(props);
  this.state = {
    file: null,
    pi: 90,
    amr: false,
    serotype: true,
    vf: true,
    submitted: false,
    open: false,
    msg: '',
    jobId: "",
    hasResult: false,
    groupresults: true,
    bulk: false,
    progress: 0
  }
}
```

and uses the `onChange` function:

```
_updateSerotype = (value) => {
  this.setState({ serotype: value })
}
```

which is appended to the form by:

```
data.append('options.serotype', this.state.serotype)
```

So if you wanted to add a new option, say `Phylotyper`, you'd create a checkbox like so:

```
<Checkbox
  id="phylotyper"
  name="check phylotyper"
  checked={phylotyper}
  onChange={this._updatePhylotyper}
```

(continues on next page)

(continued from previous page)

```
  label="Use Phylotyper"
/>
```

and add the default state as true in the constructor:

```
phylotyper: true
```

with the corresponding `onChange` function:

```
_updatePhylotyper = (value) => {
  this.setState({ phylotyper: value })
}
```

which is appended to the form by:

```
data.append('options.phylotyper', this.state.phylotyper)
```

and that's it for the form part!

## 2.8.2 Handling a New Option in ra_posts.py

Looking at the function definition, we can see that `upload()` in ra_posts.py is the route we want to edit:

```
# for Subtyping module
# the /api/v0 prefix is set to allow CORS for any postfix
# this is a modification of the old upload() methods in views.py
@bp_ra_posts.route('/api/v0/upload', methods=['POST'])
def upload():
```

We store user-selected options in the `options` dictionary defined at the beginning, with a slight exception in the `pi` option due to legacy reasons. For example, the `serotype` is defined via:

```
options['serotype']=True
```

So let's define the default for phylotyper to be true:

```
options['phylotyper']=True
```

Then we need to process the formdata. The following code block is used to convert the lower-case `false` is javascript to the upper case `False` in python, likewise with `true`:

```
# processing form data
for key, value in form.items():
    #we need to convert lower-case true/false in js to upper case in python
        #remember, we also have numbers
    if not value.isdigit():
        if value.lower() == 'false':
            value = False
        else:
            value = True
        if key == 'options.amr':
            options['amr']=value
        if key == 'options.vf':
            options['vf']=value
        if key == 'options.serotype':
```

(continues on next page)

(continued from previous page)

```
            options['serotype']=value
        if key == 'options.groupresults':
            groupresults = value
        if key == 'options.bulk':
            options['bulk'] = value
    else:
        if key =='options.pi':
            options['pi']=int(value)
```

So for `phylotyper`, we'll add an `if` block:

```
if key == 'options.phylotyper':
  options['phylotyper']=value
```

After this point, your option will be passed to the spfy.py call.

### 2.8.3 Create an enqueue() Call in spfy.py

**Warning:** A previous version of the docs recommended you create your own module (adjacent to spfy.py) to enqueue your option. Note that this is no longer recommended as you have to support the bulk uploading and the backlog option in the Subtyping.js card.

Currently, we define pipelines denoted within comment blocks:

```
# AMR PIPELINE
def amr_pipeline(multiples):
    job_amr = multiples.enqueue(amr, query_file, depends_on=job_id)
    job_amr_dict = multiples.enqueue(
        amr_to_dict, query_file + '_rgi.tsv', depends_on=job_amr)
    # this uploads result to blazegraph
    if single_dict['options']['bulk']:
        job_amr_datastruct = multiples.enqueue(
            datastruct_savvy, query_file, query_file + '_id.txt', query_file + '_rgi.
↪tsv_rgi.p', depends_on=job_amr_dict, result_ttl=-1)
    else:
        job_amr_datastruct = multiples.enqueue(
            datastruct_savvy, query_file, query_file + '_id.txt', query_file + '_rgi.
↪tsv_rgi.p', depends_on=job_amr_dict)
    d = {'job_amr': job_amr, 'job_amr_dict': job_amr_dict,
         'job_amr_datastruct': job_amr_datastruct}
    # we still check for the user-selected amr option again because
    # if it was not selected but BACKLOG_ENABLED=True, we dont have to
    # enqueue it to backlog_multiples_q since beautify doesnt upload
    # blazegraph
    if single_dict['options']['amr'] and not single_dict['options']['bulk']:
        job_amr_beautify = multiples.enqueue(
            beautify, single_dict, query_file + '_rgi.tsv_rgi.p', depends_on=job_amr_
↪dict, result_ttl=-1)
        d.update({'job_amr_beautify': job_amr_beautify})
    return d

if single_dict['options']['amr']:
    amr_jobs = amr_pipeline(multiples_q)
```

(continues on next page)

(continued from previous page)

```
    job_amr = amr_jobs['job_amr']
    job_amr_dict = amr_jobs['job_amr_dict']
    job_amr_datastruct = amr_jobs['job_amr_datastruct']
    if not single_dict['options']['bulk']:
        job_amr_beautify = amr_jobs['job_amr_beautify']
elif config.BACKLOG_ENABLED:
    amr_pipeline(backlog_multiples_q)
# END AMR PIPELINE
```

The `AMR PIPELINE` is a good reference point to start from. Note the relative imports to *app/* in *spfy.py*:

```
from modules.amr.amr import amr
```

In this case, there is an folder called `amr` with module `amr` and main method `amr`. You don't have to follow the same naming structure of course.

A simple definition for `phylotyper` might start like so:

```
def blob_savvy_enqueue(single_dict):
  # ...
  # PHYLOTYPER PIPEINE
  def phylotyper_pipeline(singles):
    # the main enqueue call
    job_phylotyper = singles.enqueue(phylotyper_main, query_file, depends_on=job_id)
    d.update('job_phylotyper': job_phylotyper)
    return d

  # check if the phylotyper option was selected by the user
  if single_dict['options']['phylotyper']:
    phylotyper_jobs = phylotyper_pipeline(singles_q)
    job_phylotyper = phylotyper_jobs['job_phylotyper']
  elif config.BACKLOG_ENABLED:
    phylotyper_pipeline(backlog_singles_q)
```

**Note:** the `singles`-type queues are used when the enqueued module can't be run in parallel on the same machine (eg. you cant open up two terminals and run the module at the same time). If the module you're adding can be run in parrallel, you can replace the `singles` queues with the `multiples` queues.

The way enqueue() works is that the first *args is the function to enqueue and the following *args are for the function itself. `depends_on` alows us to specify a job in RQ that must be completed prior to your function.

The code above is just a start and doesn't support the bulk uploading option, storing of results in blazegraph, or return to the front-end. In this case, the inner *phylotyper_pipeline()* function is used to enqueue the task. We do this to support the bulk uploading option: in the regular case where the user has selected the phylotyper option, we call the pipeline method with the `singles_q` which always runs before tasks in any `backlog_*` queue (See *Optional: Adding a new Queue* for how this is implemented). Now, if the user have enabled backlog tasks, where all tasks are run even if the user doesn't select them, then phylotyper_pipeline() is still called except:

1. We call the pipeline with the backlog queue
2. We don't care to store any job data

The additional functions: `amr_to_dict` converts the amr results into the structure required by `datastruct_savvy`. The following code-block is used to enable bulk uploading. Note that if bulk uploading is selected, we set a `result_ttl=-1` for the status checking functions in ra_statuses.py to use for checking completion.

**Note:** This `result_ttl=-1` requirement will no longer be necessary when job dependency checking is streamlined in release candidate v5.0.0

```
# this uploads result to blazegraph
if single_dict['options']['bulk']:
    job_amr_datastruct = multiples.enqueue(
        datastruct_savvy, query_file, query_file + '_id.txt', query_file + '_rgi.tsv_
↪rgi.p', depends_on=job_amr_dict, result_ttl=-1)
else:
    job_amr_datastruct = multiples.enqueue(
        datastruct_savvy, query_file, query_file + '_id.txt', query_file + '_rgi.tsv_
↪rgi.p', depends_on=job_amr_dict)
```

The `beautify` function is used to convert the return of `amr_to_dict` to the format required by the front-end React application. It is only enqueued if the `amr` option, for example, was selected but bulk uploading was not selected.

### 2.8.4 Adding a Git Submodule

**Warning:** RQ enqueus functions relative to being inside the `app/` folder, depending on your code base you may have to refactor.

The process to add a submodule for an option in the Subtyping card is the same as in *Integrating your Codebase into Spfy*. Please refer to that sectio for details.

### 2.8.5 Pickling the Result of Intermediate Tasks

We handle parsing of intermediate results by pickling the python object and storing it in the same location as the genome file. For example, amr_to_dict.py handles this by:

```
p = os.path.join(amr_file + '_rgi.p')
pickle.dump(amr_dict, open(p, 'wb'))
```

If you need to store results between tasks, do so in the same manner.

**Note:** A cleanup task will be added in release candidate v5.0.0 which wipes the temporary containing folder once all jobs are complete, so you don't have to worry about cleanup for now.

### 2.8.6 Modifying your Return to Fit datastruct_savvy.py

datastruct_savvy.py expects the format of modules which return gene hits (ex. Virulence Factors or Antimicrobial Resistance Genes) to have the form (an example of the conversion can be found in amr_to_dict.py:

```
{'Antimicrobial Resistance':
  {'somecontigid1':{'START':1, 'STOP':2, 'GENE_NAME': 'somename', 'ORIENTATION':'+',
↪'CUT_OFF':90},
  'somecontigid2':{'START':1, 'STOP':2, 'GENE_NAME': 'somename', 'ORIENTATION':'+',
↪'CUT_OFF':90},
```

(continues on next page)

(continued from previous page)

```
  'somecontigid3':{'START':1, 'STOP':2, 'GENE_NAME': 'somename', 'ORIENTATION':'+',
↪'CUT_OFF':90}
}}
```

and expects the result of serotyping as:

> **{'Serotype':** {'O-Type':'O1', 'H-Type':'H2',}
>
> }

If you were adding a return similar to `serotype`, such as with phylotyper, define a parsing function in datastruct_savvy.py similar to `parse_serotype()`:

```
def parse_serotype(graph, serotyper_dict, uriIsolate):
  if 'O type' in serotyper_dict:
      graph.add((uriIsolate, gu('ge:0001076'),
                 Literal(serotyper_dict['O type'])))
  if 'H type' in serotyper_dict:
      graph.add((uriIsolate, gu('ge:0001077'),
                 Literal(serotyper_dict['H type'])))
  if 'K type' in serotyper_dict:
      graph.add((uriIsolate, gu('ge:0001684'),
                 Literal(serotyper_dict['K type'])))

  return graph
```

Then add the call in the elif in `generate_datastruct()`:

```
# graphing functions
for key in results_dict.keys():
    if key == 'Serotype':
        graph = parse_serotype(graph,results_dict['Serotype'],uriIsolate)
    elif key == 'Virulence Factors':
        graph = parse_gene_dict(graph, results_dict['Virulence Factors'], uriGenome,
↪'VirulenceFactor')
    elif key == 'Antimicrobial Resistance':
        graph = parse_gene_dict(graph, results_dict['Antimicrobial Resistance'],␣
↪uriGenome, 'AntimicrobialResistanceGene')
return graph
```

If you're adding an option that returns specific hits, such as PanSeq, parse to results as before and call `parse_gene_dict()` on it.

```
# graphing functions
for key in results_dict.keys():
    if key == 'Serotype':
        graph = parse_serotype(graph,results_dict['Serotype'],uriIsolate)
    elif key == 'Virulence Factors':
        graph = parse_gene_dict(graph, results_dict['Virulence Factors'], uriGenome,
↪'VirulenceFactor')
    elif key == 'Antimicrobial Resistance':
        graph = parse_gene_dict(graph, results_dict['Antimicrobial Resistance'],␣
↪uriGenome, 'AntimicrobialResistanceGene')
    elif key == 'Panseq':
        graph = parse_gene_dict(graph, results_dict['Panseq'], uriGenome,
↪'PanseqRegion')
return graph
```

You'll then have to enqueue the `datastruct_savvy()` call in spfy.py similar to:

```
# this uploads result to blazegraph
if single_dict['options']['bulk']:
    job_amr_datastruct = multiples.enqueue(
        datastruct_savvy, query_file, query_file + '_id.txt', query_file + '_rgi.tsv_
↪rgi.p', depends_on=job_amr_dict, result_ttl=-1)
else:
    job_amr_datastruct = multiples.enqueue(
        datastruct_savvy, query_file, query_file + '_id.txt', query_file + '_rgi.tsv_
↪rgi.p', depends_on=job_amr_dict)
```

Then the datastruct result is added to the *d* dictionary of your inner pipeline function:

```
d = {'job_amr': job_amr, 'job_amr_dict': job_amr_dict,
     'job_amr_datastruct': job_amr_datastruct}
```

and, outside of the inner function, it's assigned as `job_amr_datastruct`:

```
job_amr_datastruct = amr_jobs['job_amr_datastruct']
```

By default, we set the datastruct as the end task to send back - this is to faciliate bulk uploading. If the user-doesn't select the bulk option, then the return is the result from `beautify()`:

```
# new to 4.3.3 if bulk ids used return the endpoint of datastruct generation
# to poll for completion of all jobs
# these two ifs handle the case where amr (or vf or serotype) might not
# be selected but bulk is
if (single_dict['options']['vf'] or single_dict['options']['serotype']):
    ret_job_ectyper = job_ectyper_datastruct
if single_dict['options']['amr']:
    ret_job_amr = job_amr_datastruct
# if bulk uploading isnt used, return the beautify result as the final task
if not single_dict['options']['bulk']:
    if (single_dict['options']['vf'] or single_dict['options']['serotype']):
        ret_job_ectyper = job_ectyper_beautify
    if single_dict['options']['amr']:
        ret_job_amr = job_amr_beautify
# add the jobs to the return
if (single_dict['options']['vf'] or single_dict['options']['serotype']):
    jobs[ret_job_ectyper.get_id()] = {'file': single_dict[
        'i'], 'analysis': 'Virulence Factors and Serotype'}
if single_dict['options']['amr']:
    jobs[ret_job_amr.get_id()] = {'file': single_dict[
        'i'], 'analysis': 'Antimicrobial Resistance'}
```

### 2.8.7 Modifying beautify.py

Technically, you'll mostly be using the `json_return()` method from beautify.py as it performs the core conversion to json. `beautify()` also performs a number of checks that are specific to ECTyper and RGI: namely, we parse the `gene_dict` and find the widest hit in a given contig. For new modules, we recommand you just create a basic function in beautify.py to perform the `pickle.load()` to bypass the widest_hit search and failed handling. For example:

```
def beautify_myoption(args_dict, pickled_dictionary):
  gene_dict = pickle.load(open(pickled_dictionary, 'rb'))
```

(continues on next page)

(continued from previous page)

```
  # this converts our dictionary structure into json and adds metadata (filename, etc.
↪)
  json_r =  json_return(args_dict, gene_dict)
  return json_r
```

If you're adding a serotyping tool such as `phylotyper`, modifying:

```
if analysis == 'Serotype':
```

to be:

```
if analysis in ('Serotype','Phylotyper'):
```

should be all the modification to `json_return()` that is required.

For results similar to VF/AMR, where we have a list of genes, you can call `json_return()` directly without modification.

With beautify.py modified, add the `beautify_myoption()` call to your pipeline like so:

```
if single_dict['options']['phylotyper'] and not single_dict['options']['bulk']:
  job_phylotyper_beautify = multiples.enqueue(
      beautify_myoption, single_dict, query_file + '_phylotyper.p', depends_on=job_
↪phylotyper_dict, result_ttl=-1)
  d.update({'job_phylotyper_beautify': job_phylotyper_beautify})
```

and then set the result as the return to the front-end:

```
# if bulk uploading isnt used, return the beautify result as the final task
if not single_dict['options']['bulk']:
    if (single_dict['options']['vf'] or single_dict['options']['serotype']):
        ret_job_ectyper = job_ectyper_beautify
    if single_dict['options']['amr']:
        ret_job_amr = job_amr_beautify
```

## 2.9 Debugging

You can see all the containers on your host computer by running:

```
docker ps
```

When running commands within `/backend` (at the same location as the `docker-compose.yml` file), you can see the composition-specific containers by running:

```
docker-compose logs
```

Within the repo, you can also see logs for specific containers by referencing the service name, as defined in the `docker-compose.yml` file. For example, logs for the Flask webserver can be retrieved by running:

```
docker-compose logs webserver
```

or if you wanted the tail:

```
docker-compose logs --tail=100 webserver
```

or for Blazegraph:

```
docker-compose logs blazegraph
```

To clean up after Docker, see the excellent Digital Ocean guide on How To Remove Docker Images, Containers, and Volumes.

### 2.9.1 Monitoring Flask

Three options:

1. Docker captures all *stdout* messages into Docker's logs. You can see them by running:

   ```
   docker logs backend_webserver_1
   ```

2. Flask is also configured to report errors via Sentry; copy your DSN key and uncomment the `SENTRY_DSN` option in `/app/config.py`.
3. Drop a shell info the webserver container, then you can run explore the file structure from there. The webserver will typically run as `backend_webserver_1`. Note that there won't be any `access.log` or similar as this information is collected through Docker's logs.

### 2.9.2 Monitoring RQ

To monitor the status of RQ tasks and check on failed jobs, you have two options:

1. Setup a https://sentry.io account and copy your DSN into `/app/config.py`
2. Port 9181 is mapped to host on Service `backend-rq`, you can use `rq-dashboard` via:

1. `docker exec -it backend_worker_1 sh` this drops a shell into the rq worker container which has rq-dashboard installed via conda
2. `rq-dashboard -H redis` runs rq-dashboard and specifies the *redis* host automatically defined by docker-compose
3. then on your host machine visit http://localhost:9181

We recommend using `RQ-dashboard` to see jobs being enqueued live when testing as `Sentry` only reports failed jobs. On remote deployments, we use `Sentry` for error reporting.

**Warning:** `RQ-dashboard` will not report errors from the Flask webserver. In addition, jobs enqueued with `depends_on` will not appear on the queues list until their dependencies are complete.

### 2.9.3 Debugging Javascript

For testing simple commands, I use the Node interpreter similar to how one might use Python's interpreter:

```
node
.exit
```

We use the Chrome extension React Dev Tools to see our components and state, as defined in React; Chrome's DevTools will list `Elements` in their HTML form which, while not particularly useful to debug React-specific code, can be used to check which CSS stylings are applied.

The Redux Dev Tools extension is used to monitor the state of our reactapp's Redux store. This is useful to see that your `jobs` are added correctly.

Finally, if you clone our reactapp repo, and run:

```
yarn start
```

any saved changes will be linted with `eslint`.

## 2.10 Editing the Docs

### 2.10.1 Setup

```
cd docs/
sphinx-autobuild source _build_html
```

Then you can visit http://localhost:8000 to see you changes live. Note that it uses the default python theme locally, and the default readthedocs theme when pushed.

CHAPTER 3

# Deplyoment Guide

**Table of Contents**

The way we recommend you deploy Spfy is to simply use the Docker composition for everything; this approach is documented in *Deploying in General*. Specifics related to the NML's deployment is given in *Deploying to Corefacility*.

## 3.1 Deploying in General

Most comments are based of the `docker-compose.yml` file at the project root.

### 3.1.1 Host to Container Mapping

There are a few key points to note:

```
ports:
- "8000:80"
```

The configuration maps `host:container`; so port 8000 on the host (your computer) is linked to port 80 of the container. Fields like volumes typically have only one value: `/var/lib/jetty/`; this is done to instruct Docker to map the folder `/var/lib/jetty` within the container itself to a generic volume managed by Docker, thereby enabling the data to persist across start/stop cycles.

You can also add a host path to volume mappings such as `/dbbackup/:/var/lib/jetty/` so that Docker uses an actual path on your host, instead of a generic Docker-managed volume. As before, the first term, `/dbbackup/` would reside on the host.

**Warning:** Generally, you should stop a Docker composition by running `docker-compose stop` instead of `docker-compose down`. As of the most recent Docker versions, a `docker-compose down` should not remove the Docker volumes, but this has been inconsistent in the past.

### 3.1.2 Volume Mapping in Production

In production, at minimum we recommend you map Blazegraph's volume to a backup directory. `/datastore` also stores all the uploaded genome files and related temporary files generated during analysis.

### 3.1.3 Ports

`grouch` is the front-end user interface for Spfy whereas `webserver` serves the backend Flask APIs. Without modification, when you run `docker-compose up` port 8090 is used to access the app. The front-end then calls port 8000 to submit requests to the backend. This approach is fine for individual users on their own computer, but this setup should not be used for production as it would you would have to open a separate port for api calls to be made to.

Instead, we recommend you change the port for `grouch` to the standard port 80, map the `webserver` to a subdomain, and use a reverse-proxy to resolve the subdomain to an internal port. For example, lets say you have a static ip of 137.122.64.157 and a web domain of spfy.ca . You have an A Record that maps spfy.ca to 137.122.64.157. You could then expose port 80 externally on your host and map `grouch` to port 80 by setting:

```
ports:
- "80:3000"
```

Port 8000 for `webserver` will still be available on your hosts loopback, but will not be exposed externally. Add an A Record for api.spfy.ca to the same IP address, and then you could use an reverse-proxy such as Nginx to resolve api.spfy.ca to localhost:8000.

### 3.1.4 Setting a Subdomain

This has to be done through the interface of your domain registrar. You'll have to add an Address Record (A Record), which is typically under the heading "Manage Advanced DNS Records" or similar.

### 3.1.5 Setting up a Reverse Proxy

We recommend you use NGINX as the reverse proxy. You can find their Getting Started guide at https://www.nginx.com/resources/wiki/start/

In addition, we recommend you use Certbot (part of the EFF's Let's Encrypt) project to get the required certificates and setup HTTPS on your server. You can find their interactive guide at https://certbot.eff.org/ which allow's you to specify the webserver (NGINX) and operating system you are using. Certbot comes with a nice script to automatically modify your NGINX configuration as required.

### 3.1.6 Point Reactapp to Your Subdomain

To tell reactapp to point to your subdomain, you'll have to modify the `api.js` settings located at `reactapp/src/middleware/api.js`.

The current `ROOT` of the target domain is:

```
const ROOT = window.location.protocol + '//' + window.location.hostname + ':8000/'
```

change this to:

```
const ROOT = 'https' + '//' + 'api.mydomain.com' + '/'
```

and then rebuild and redeploy reactapp.

```
docker-compose build --no-cache reactapp
docker-compose up -d
```

**e** The Flask webserver has Cross-Origin Requests (CORS) enabled, so you can deploy reactapp to another server (that is only running reactapp, and not the webserver, databases, workers). The domain can be `mydomain.com` or any domain name you own - you'll just have to setup the A records as appropriate.

## 3.2 Deploying to Corefacility

### 3.2.1 Quick-Start

Use the `production.sh` script. This script does a few things:

1. Stops the host Nginx so Docker can bind the ports it'll need for mapping.
2. Starts the Docker-Composition.
3. Restarts the host Nginx.
4. Starts Jetty which runs Blazegraph.

### 3.2.2 Important Volumes

The `webserver` Docker container has a `/datastore` directory with all submitted files.

The `mongodb` Docker container has a `/data/db` directory which persists the `Genome File Hash : SpfyID` mapping. (As well the `?token=` user sessions).

If you accidentally delete the MongoDB volume, it can be incrementally (when the same file is submitted, it will be re-cached) recreated from Blazegraph by setting `DATABASE_EXISTING = True` and `DATABASE_BYPASS = False` in `app/config.py`.

### 3.2.3 Merging New Changes Into Production

The production-specific changes are committed to the local git history in corefacility.

Running:

```
git merge origin/somebranch
```

will be sufficient to merge.

We can then rebuild and restart the composition:

```
docker-compose build --no-cache
./production.sh
```

### 3.2.4 Blazegraph

Looking at the filesystem:

```
[claing@superphy backend-4.3.3]$ df -h
Filesystem                   Size  Used Avail Use% Mounted on
/dev/mapper/superphy-root     45G   31G   14G  69% /
devtmpfs                      12G     0   12G   0% /dev
tmpfs                         12G  2.5G  9.3G  21% /dev/shm
tmpfs                         12G   26M   12G   1% /run
tmpfs                         12G     0   12G   0% /sys/fs/cgroup
/dev/vda1                    497M  240M  258M  49% /boot
/dev/mapper/docker-docker    200G   21G  180G  11% /docker
warehouse:/ifs/Warehouse     769T  601T  151T  81% /Warehouse
tmpfs                        2.4G     0  2.4G   0% /run/user/40151
tmpfs                        2.4G     0  2.4G   0% /run/user/40290
```

`/Warehouse` is used for long-term data storage and shared across the NML. In order to write to `/Warehouse`, you need the permissions of either `claing` or `superphy`; there are some problems with passing these permissions into Docker environments, so we run Blazegraph, inside of folder `/Warehouse/Users/claing/superphy/spfy/docker-blazegraph/2.1.4-inferencing` and as `claing`, outside of Docker using a jetty server.

See https://github.com/superphy/backend/issues/159

### 3.2.5 Docker Service

```
[claing@superphy docker]$ sudo cat /etc/fstab

#
# /etc/fstab
# Created by anaconda on Thu Dec 24 17:40:08 2015
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/superphy-root /                       xfs     defaults        1 1
UUID=6c62e5cf-fd55-41e8-8122-e5e78643e3cd /boot                   xfs     defaults    ␣
↪     1 2
/dev/mapper/superphy-swap swap                    swap    defaults        0 0
warehouse:/ifs/Warehouse        /Warehouse      nfs     defaults        0 0
/dev/mapper/docker-docker /docker xfs defaults 1 2
```

Our root filesystem for the Corefacility VM is really small (45G) and we instead have a virtual drive at `/dev/mapper/docker-docker` which is mounted on `/docker` which has our Docker images / unmapped volumes. This is setup using symlinks:

```
sudo systemctl stop docker
cd /var/lib/
sudo cp -rf docker/ /docker/backups/
sudo rm -rf docker/
sudo mkdir /docker/docker
sudo ln -s /docker/docker /var/lib/docker
sudo systemctl start docker
```

### 3.2.6 Docker Hub

Docker Hub is used to host pre-built images; for us, this mostly consisting of our base `docker-flask-conda` image. The org. page is publically available at https://hub.docker.com/u/superphy/ and you can pull without any permission issues. To push a new image, first register an account at https://hub.docker.com/

The owner for the org. has the username `superphyinfo` and uses the same password as `superphy.info@gmail.com`. You can use it to add yourself to the org.

You can then build and tag docker images to be pushed onto Docker Hub.

```
docker build -f Dockerfile-reactapp -t superphy/reactapp:4.3.3-corefacility .
```

or tag an existing image:

```
docker images
docker tag 245d7e4bb63e superphy/reactapp:4.3.3-corefacility
```

Either way, you can then push using the same command:

```
docker push superphy/reactapp:4.3.3-corefacility
```

**Note:** We occasionally use Docker Hub as a work-around in case a computer can't build an image. There is some bug where Corefacility VMs aren't connecting to NPM and thus we build the reactapp image on Cybera and pull it down on Corefacility.

### 3.2.7 Nginx

We run Nginx above the Docker layer for 3 reasons:

1. Handle the `/superphy` prefix to all our routes as we don't sure on /
2. To host both the original SuperPhy and Spfy on a single VM
3. Buffer large file uploads before sending it to Spfy's Flask API

In `/etc/nginx/nginx.conf`:

```
# For more information on configuration, see:
#   * Official English Documentation: http://nginx.org/en/docs/
#   * Official Russian Documentation: http://nginx.org/ru/docs/

user spfy;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /run/nginx.pid;

# Load dynamic modules. See /usr/share/nginx/README.dynamic.
include /usr/share/nginx/modules/*.conf;

events {
        worker_connections 1024;
}

http {
        log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                              '$status $body_bytes_sent "$http_referer" '
                              '"$http_user_agent" "$http_x_forwarded_for"';

        access_log  /var/log/nginx/access.log  main;
        error_log /var/log/nginx/error.log warn;

        sendfile            on;
        tcp_nopush          on;
        tcp_nodelay         on;
        keepalive_timeout   2m;
        types_hash_max_size 2048;

        include             /etc/nginx/mime.types;
        default_type        application/octet-stream;

        # Load modular configuration files from the /etc/nginx/conf.d directory.
        # See http://nginx.org/en/docs/ngx_core_module.html#include
        # for more information.
        include /etc/nginx/conf.d/*.conf;

        map $http_upgrade $connection_upgrade {
                default upgrade;
                ''      close;
        }

        server {
        client_max_body_size 60g;
        listen       80 default_server;
        listen       443 ssl http2 default_server;
```

(continues on next page)

(continued from previous page)

```
                listen       [::]:80 default_server;
        listen       [::]:443 ssl http2 default_server;
        server_name  superphy.corefacility.ca;
                # Load configuration files for the default server block.
                include /etc/nginx/default.d/*.conf;


        location / {
                        proxy_pass http://127.0.0.1:8081;
        }
                location /grouch {
                        return 301 /superphy/spfy/;
                }
        location /superphy/grouch {
                        return 301 /superphy/spfy/;
                }
        location /spfyapi/ {
                rewrite ^/spfyapi/(.*)$ /$1 break;
                        proxy_pass http://localhost:8090;
                        proxy_redirect http://localhost:8090/ $scheme://$host/spfyapi/
↪;
                        proxy_http_version 1.1;
                        proxy_set_header Upgrade $http_upgrade;
                        proxy_set_header Connection $connection_upgrade;
                        proxy_read_timeout 20d;
        }
        location /spfy/ {
                        rewrite ^/spfy/(.*)$ /$1 break;
                        proxy_pass http://localhost:8091;
                        proxy_redirect http://localhost:8091/ $scheme://$host/spfy/;
                        proxy_http_version 1.1;
                        proxy_set_header Upgrade $http_upgrade;
                        proxy_set_header Connection $connection_upgrade;
                        proxy_read_timeout 20d;
                }
        location /shiny/ {
                rewrite ^/shiny/(.*)$ /$1 break;
                proxy_pass http://127.0.0.1:3838;
                proxy_redirect http://127.0.0.1:3838/ $scheme://$host/shiny/;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection $connection_upgrade;
                proxy_read_timeout 950s;
        }

        }

        server {
                client_max_body_size 60g;
                listen       80;
                listen       443 ssl http2;
                listen       [::]:80;
                listen       [::]:443 ssl http2;
                server_name  lfz.corefacility.ca;
                # Load configuration files for the default server block.
                include /etc/nginx/default.d/*.conf;
```

(continues on next page)

(continued from previous page)

```
        location / {
                        proxy_pass http://127.0.0.1:8081;
        }
        location = /spfy {
                return 301 /superphy/spfy/;
        }
        location /grouch {
                        return 301 /superphy/spfy/;
                }
                location /superphy/grouch {
                        return 301 /superphy/spfy/;
                }
                location = /minio {
                        return 301 /superphy/minio/;
                }
        location /spfyapi/ {
                        rewrite ^/spfyapi/(.*)$ /$1 break;
                proxy_pass http://localhost:8090;
                        proxy_redirect http://localhost:8090/superphy/ $scheme://
↪$host/spfyapi/;
                        proxy_http_version 1.1;
                        proxy_set_header Upgrade $http_upgrade;
                        proxy_set_header Connection $connection_upgrade;
                        proxy_read_timeout 20d;
                }
        location /spfy/ {
                        #rewrite ^/spfy/(.*)$ https://superphy.github.io/status/␣
↪redirect;
                rewrite ^/spfy/(.*)$ /$1 break;
                        proxy_pass http://localhost:8091;
                        proxy_redirect http://localhost:8091/superphy/ $scheme://
↪$host/spfy/;
                        proxy_http_version 1.1;
                        proxy_set_header Upgrade $http_upgrade;
                        proxy_set_header Connection $connection_upgrade;
                        proxy_read_timeout 2h;
                proxy_send_timeout 2h;
                }
        location /minio/ {
                        rewrite ^/minio/(.*)$ /$1 break;
                        proxy_pass http://localhost:9000;
                        proxy_redirect http://localhost:9000/superphy/ $scheme://
↪$host/minio/;
                        proxy_http_version 1.1;
                        proxy_set_header Upgrade $http_upgrade;
                        proxy_set_header Connection $connection_upgrade;
                        proxy_read_timeout 2h;
                        proxy_send_timeout 2h;
                }
        location /shiny/ {
                rewrite ^/shiny/(.*)$ /$1 break;
                        proxy_pass http://127.0.0.1:3838;
                        proxy_redirect http://127.0.0.1:3838/ $scheme://$host/shiny/;
                        proxy_http_version 1.1;
                        proxy_set_header Upgrade $http_upgrade;
                        proxy_set_header Connection $connection_upgrade;
                proxy_read_timeout 950s;
```

(continues on next page)

(continued from previous page)

```
            }
            }



}
```

This is setup to run the ReactJS frontend of Spfy (`grouch`) at https://lfz.corefacility.ca/superphy/spfy/ and the api at https://lfz.corefacility.ca/superphy/spfyapi/

Points to note:

- The rewrite rules are critical to operating on Corefacility, as the `/superphy/` requirement can be tricky
- We're unsure if the `client_max_body_size 60g;` has any effect when deployed on Corefacility, it might be that there is another Nginx instance ran by the NML to route its VMs. Currently we're capped at ~250 MB uploads at a time on Corefacility, you can see a long debugging log of this at https://github.com/superphy/backend/issues/159
- Nginx is not hosting the websites, it only serves to proxy the requests to Apache (for the old SuperPhy) or Docker (for the new Spfy)

**Warning:** Nginx is also run internally in the Docker webserver image to allow you to handle running the composition by itself, but generally you shouldn't have to worry about it.

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search