sfm Documentation

Release 0.6.1

The George Washington University Libraries

June 02, 2016

Contents

Social Feed Manager is open source software for libraries, archives, cultural heritage institutions and research organizations. It empowers those communities' researchers, faculty, students, and archivists to define and create collections of data from social media platforms. Social Feed Manager will harvest from Twitter, Tumblr, Flickr, and Sina Weibo and is extensible for other platforms. In addition to collecting data from those platforms' APIs, it will collect linked web pages and media.

This site provides documentation for installation and usage of SFM. See the Social Feed Manager project site for full information about the project's objectives, roadmap, and updates.

CHAPTER 1

Contents:

1.1 Installation and configuration

1.1.1 Overview

The supported approach for deploying SFM is Docker containers.

Each SFM service will provide images for the containers needed to run the service (in the form of `Dockerfile`s). These images will be published to Docker Hub. GWU created images will be part of the GWUL organization and be prefixed with *sfm-*.

sfm-docker provides the necessary `docker-compose.yml` files to compose the services into a complete instance of SFM.

For a container, there may be multiple flavors of the container. In particular, there may be the following:

- *development*: The code for the service is outside the container and linked into the container as a shared volume. This supports development with a running instance of the service.
- *master*: The container contains the master branch of the code at the time the image is built.
- *release*: The container contains a release of the code. There will be a separate image for each release.

For more information, see Docker.

SFM *can* be deployed without Docker. The various ‘‘Dockerfile‘‘s should provide reasonable guidance on how to accomplish this.

1.1.2 Configuration

- Passwords are kept in `secrets.env`. A template for this file (`example.secrets.env`) is provided.
- Debug mode for sfm-ui is controlled by the `DEBUG` environment variable in `docker-compose.yml`. If setting `DEBUG` to false, the `SFM_HOST` environment variable must be provided with the host. See the Django documentation for `ALLOWED_HOSTS`.
- The default timezone is Eastern Standard Time (EST). To select a different timezone, change `TZ=EST` in `docker-compose.yml`.
- Email is configured by providing the `SFM_HOST`, `SFM_SMTP_HOST`, `SFM_EMAIL_USER`, and `SFM_EMAIL_PASSWORD` environment variables. `SFM_HOST` is used to determine the host name when constructing links contained in the emails.

- Application credentials for social media APIs are configured by providing the `TWITTER_CONSUMER_KEY`, `TWITTER_CONSUMER_SECRET`, `WEIBO_API_KEY`, and/or `WEIBO_API_SECRET`. For more information, see API Credentials.
- The data volume strategy is used to manage the volumes that store SFM's data. By default, normal Docker volumes are used; to use a host volume instead, add the host directory to the `volumes` field. This will allow you to access the data outside of Docker. For example:

```
    sfmdata:
        image: ubuntu:14.04
        command: /bin/true
        volumes:
             - /myhost/data:/sfm-data
```

1.1.3 Local installation

Installing locally required Docker and Docker-Compose. See *Installing Docker*.

1. Either clone this repository:

```
    git clone git@github.com:gwu-libraries/sfm-docker.git
```

or just download `docker-compose.yml` and `example.secrets.env`:

```
curl -L https://github.com/gwu-libraries/sfm-docker/raw/master/master.docker-compose.yml > docker-com
curl -L https://github.com/gwu-libraries/sfm-docker/raw/master/example.secrets.env > secrets.env
```

2. Put real secrets in `secrets.env`.
3. Bring up the containers:

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

1.1.4 Amazon EC2 installation

To launch an Amazon EC2 instance running SFM, follow the normal procedure for launching an instance. In *Step 3: Configure Instance Details*, under *Advanced Details* paste the following in user details and modify as appropriate:

```
#cloud-config
repo_update: true
repo_upgrade: all

packages:
 - python-pip

runcmd:
 - curl -sSL https://get.docker.com/ | sh
 - usermod -aG docker ubuntu
 - pip install -U docker-compose
 - mkdir /sfm-data
# This brings up master. To bring up a specific version, replace master with the
# version number, e.g., 0.6.0.
 - curl -L https://github.com/gwu-libraries/sfm-docker/raw/master/master.docker-compose.yml > docker-
 - curl -L https://github.com/gwu-libraries/sfm-docker/raw/master/example.secrets.env > secrets.env
# Set secrets below. Secrets that are not commented out are required.
# Secrets that are commented out are not required. To include, remove the #.
# Don't forget to escape $ as \$.
```

```
# The password used for logging into the Rabbit Admin. Username is sfm_user.
 - echo RABBITMQ_DEFAULT_PASS=password >> secrets.env
# Postgres password.
 - echo POSTGRES_PASSWORD=password >> secrets.env
# The password for the admin account for SFM UI. Username is sfmadmin.
 - echo SFM_SITE_ADMIN_PASSWORD=password >> secrets.env
# The account used to send email via SMTP from SFM UI.
# - echo SFM_EMAIL_USER=justinlittman@email.gwu.edu >> secrets.env
# - echo SFM_EMAIL_PASSWORD=password >> secrets.env
# The password used to log into the Heritrix UI. Username is sfm_user.
 - echo HERITRIX_PASSWORD=password >> secrets.env
# API keys for allowing users to connect to social media platform APIs.
# If not provided, credentials can still be provided in SFM UI.
# - echo TWITTER_CONSUMER_KEY=EHdoeW7ksBgflP5nUalEfhao >> secrets.env
# - echo TWITTER_CONSUMER_SECRET=ZtUemftBkf2cEmaqiyW2Ddihu9FPAiLebuMOmqN0jeQtXeAlen >> secrets.env
# - echo WEIBO_API_KEY=1313340598 >> secrets.env
# - echo WEIBO_API_SECRET=68ae6a497f2f6eac07ec14bf7c0e0fa52 >> secrets.env
# Values must be provided for all of the following.
# HERITRIX_CONTACT_URL is included in the HTTP request when harvesting web
# resources with Heritrix.
 - export HERITRIX_CONTACT_URL=http://library.gwu.edu
# The following are optional.
# The SMTP server used to send email.
 - export SMTP_HOST=smtp.gmail.com
# The email address of the admin account for SFM UI.
 - export SITE_ADMIN_EMAIL=nowhere@example.com
# The time zone.
 - export TZ=EST
# The host name of the server.
 - export HOST=`curl http://169.254.169.254/latest/meta-data/public-hostname`
 - sed -i 's/\/sfm-data/"\/sfm-data:\/sfm-data"/' docker-compose.yml
 - sed -i "s/HERITRIX_CONTACT_URL=http:\/\/library.gwu.edu/HERITRIX_CONTACT_URL=${HERITRIX_CONTACT_UR
 - sed -i "s/SFM_SMTP_HOST=smtp.gmail.com/SFM_SMTP_HOST=${SMTP_HOST}/" docker-compose.yml
 - sed -i "s/SFM_SITE_ADMIN_EMAIL=nowhere@example.com/SFM_SITE_ADMIN_EMAIL=${SITE_ADMIN_EMAIL}/" doc
 - sed -i "s/TZ=EST/TZ=${TZ}/g" docker-compose.yml
 - sed -i "s/SFM_HOST=sfm.gwu.edu:8080/SFM_HOST=${HOST}/" docker-compose.yml
 - docker-compose up -d
```

When the instance is launched, SFM will be installed and started.

Note the following:

- Starting up the EC2 instance will take several minutes.
- This has been tested with *Ubuntu Server 14.04 LTS*, but may work with other AMI types.
- We don't have recommendations for sizing, but providing multiple processors even for testing/experimentation.
- If you need to make additional changes to your `docker-compose.yml`, you can ssh into the EC2 instance and make changes. `docker-compose.yml` and `secrets.env` will be in the default user's home directory.
- Make sure to configure a security group that exposes the proper ports. To see which ports are used by which services, see master.docker-compose.yml.
- To learn more about configuring EC2 instances with user data, see the AWS user guide.

1.2 Authentication

Social Feed Manager allows users to self-sign up for accounts. Those accounts are stored and managed by SFM. Future versions of SFM will support authentication against external systems, e.g., Shibboleth.

By default, a group is created for each user and the user is placed in group. To create additional groups and modify group membership use the Admin interface.

In general, users and groups can be administered from the Admin interface.

The current version of SFM is not very secure. Future versions of SFM will more tightly restrict what actions users can perform and what they can view. In the meantime, it is encouraged to take other measures to secure SFM such as restricting access to the IP range of your institution.

1.3 API Credentials

Accessing the APIs of social media platforms requires credentials for authentication (also knows as API keys). Social Feed Manager supports managing those credentials.

Most API credentials have two parts: an application credential and a user credential. (Flickr is the exception – only an application credential is necessary.)

It is important to understand how credentials/authentication effect what API methods can be invoked and rate limits. For more information, consult the documentation for each social media platform's API.

1.3.1 Managing credentials

SFM supports two approaches to managing credentials: adding credentials and connecting credentials. Both of these options are available from the Credentials page.

Adding credentials

For this approach, a user gets the application and/or user credential from the social media platform and provide them to SFM by completing a form. More information on getting credentials is below.

Connecting credentials

For this approach, SFM is configured with the application credentials for the social media platform. The user credentials are obtained by the user being redirected to the social media website to give permission to SFM to access her account.

SFM is configured with the application credentials in the `docker-compose.yml`. If additional management is necessary, it can be performed using the Social Accounts section of the Admin interface.

This is the easiest approach for users. Configuring application credentials is encouraged.

1.3.2 Platform specifics

Twitter

Twitter credentials can be obtained from https://apps.twitter.com/.

Weibo

For instructions on obtaining Weibo credentials, see this guide.

To use the connecting credentials approach for Weibo, the redirect URL must match the application's actual URL and use port 80.

Flickr

Flickr credentials can be obtained from https://www.flickr.com/services/api/keys/.

Flickr does not require user credentials.

1.4 Processing

Your social media data can be used in a processing/analysis pipeline. SFM provides several tools and approaches to support this.

1.4.1 Tools

Warc iterators

A warc iterator tool provides an iterator to the social media data contained in WARC files. When used from the commandline, it writes out the social items one at a time to standard out. (Think of this as `cat`-ing a line-oriented JSON file. It is also equivalent to the output of Twarc.)

Each social media type has a separate warc iterator tool. For example, `twitter_rest_warc_iter.py` extracts tweets recorded from the Twitter REST API. For example:

```
root@0ac9caaf7e72:/sfm-data# twitter_rest_warc_iter.py
usage: twitter_rest_warc_iter.py [-h] [--pretty] [--dedupe]
                                 [--print-item-type]
                                 filepaths [filepaths ...]
```

Warc iterator tools can also be used as a library.

Find Warcs

`find_warcs.py` helps put together a list of WARC files to be processed by other tools, e.g., warc iterator tools. (It gets the list of WARC files by querying the SFM API.)

Here is arguments it accepts:

```
root@0ac9caaf7e72:/sfm-data# find_warcs.py
usage: find_warcs.py [-h] [--include-web] [--harvest-start HARVEST_START]
                     [--harvest-end HARVEST_END] [--api-base-url API_BASE_URL]
                     [--debug [DEBUG]]
                     collection [collection ...]
```

For example, to get a list of the WARC files in a particular collection, provide some part of the collection id:

```
root@0ac9caaf7e72:/sfm-data# find_warcs.py 4f4d1
/sfm-data/collections/b06d164c632d405294d3c17584f03278/4f4d1a6677f34d539bbd8486e22de33b/2016/05/04/14
```

(In this case there is only one WARC file. If there was more than one, it would be space separated.)

The collection id can be found from the SFM UI.

Note that if you are running `find_warcs.py` from outside a Docker environment, you will need to supply `--api-base-url`.

1.4.2 Approaches

Processing in container

To bootstrap processing, a processing image is provided. A container instantiated from this image is Ubuntu 14.04 and pre-installed with the warc iterator tools, `find_warcs.py`, jq, Twarc (for access to the Twarc utilities). It will also have access to the data from `/sfm-data/collections`.

To instantiate:

```
docker run -it --rm --link=docker_sfmapp_1:api --volumes-from=docker_sfmdata_1 gwul/sfm-processing:0.
```

The arguments will need to be adjusted depending on your Docker environment.

You will then be provided with a bash shell inside the container from which you can execute commands:

```
root@0ac9caaf7e72:/sfm-data# find_warcs.py 4f4d1 | xargs twitter_rest_warc_iter.py | python /opt/twar
```

Processing locally

In a typical Docker configuration, the data directory will be linked into the Docker environment. This means that the data is available both inside and outside the Docker environment. Given this, processing can be performed locally (i.e., outside of Docker).

The various tools can be installed locally:

```
GLSS-F0G5RP:tmp justinlittman$ virtualenv ENV
GLSS-F0G5RP:tmp justinlittman$ source ENV/bin/activate
(ENV)GLSS-F0G5RP:tmp justinlittman$ pip install git+https://github.com/gwu-libraries/sfm-utils.git
(ENV)GLSS-F0G5RP:tmp justinlittman$ pip install git+https://github.com/gwu-libraries/sfm-twitter-harv
(ENV)GLSS-F0G5RP:tmp justinlittman$ twitter_rest_warc_iter.py
usage: twitter_rest_warc_iter.py [-h] [--pretty] [--dedupe]
                                 [--print-item-type]
                                 filepaths [filepaths ...]
twitter_rest_warc_iter.py: error: too few arguments
```

1.5 Development

1.5.1 Setting up a development environment

SFM is composed of a number of components. Development can be performed on each of the components separately. The following describes setting up an development environment for a component.

Step 1: Pick a development configuration

For SFM development, it is recommended to run components within a Docker environment (instead of directly in your OS, not in Docker). Docker runs natively (and cleanly) on Ubuntu; on OS X Docker requires Docker Toolbox.

Since Docker can't run natively on OS X, Docker Toolbox runs it inside a VirtualBox VM, which is largely transparent to the user. Note that GWU's configuration of the Cisco AnyConnect VPN client breaks Docker Toolbox. You can work around this with vpn_fix.sh, but this is less than optimal.

Depending on your development preferences and the OS you development on, you may want to consider one of the following configurations:

- Develop locally and run Docker locally: Optimal if using an IDE and not using OS X/ Cisco AnyConnect.
- Both develop and run Docker in an Ubuntu VM. The VM can be local (e.g., in VMWare Fusion) or remote Ubuntu VM (e.g., a WRLC or AWS VM): Optimal if using a text editor.
- Develop locally and run Docker in a local VM with the local code shared into the VM: Optimal if using an IDE.

Step 2: Install Docker and Docker Compose

See See *Installing Docker*.

Step 3: Clone the component's repo

For example:

```
git clone https://github.com/gwu-libraries/sfm-ui.git
```

Step 4: Configure *docker-compose.yml*

Each SFM component should provide a development Docker image and an example *dev.docker-compose.yml* file (in the *docker/* directory).

The development Docker image will run the component using code that is shared with container. That is, the code is made available at container run time, rather than build time (as it is for master or production images). This allows you to change code and have it affect the running component if the component (e.g., a Django application) is aware of code changes. If the component is not aware of code changes, you will need to restart the container to get the changes (*docker kill <container name>* followed by *docker-compose up -d*).

The development *docker-compose.yml* will bring up a container running the component and containers for any additional components that the component depends on (e.g., a RabbitMQ instance). Copy *dev.docker-compose.yml* to *docker-compose.yml* and update it as necessary. At the very least, you will need to change the volumes link to point to your code:

```
volumes:
    - "<path of your code>:/opt/sfm-ui"
```

You may also need to change the defaults for exposed ports to ports that are available in your environment.

Step 5: Run the code

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

For additional Docker and Docker-Compose commands, see below.

1.5.2 Development tips

Admin user accounts

When running a development *docker-compose.yml*, each component should automatically create any necessary admin accounts (e.g., a django admin for SFM UI). Check *dev.docker-compose.yml* for the username/passwords for those accounts.

RabbitMQ management console

The RabbitMQ management console can be used to monitor the exchange of messages. In particular, use it to monitor the messages that a component sends, create a new queue, bind that queue to *sfm_exchange* using an appropriate routing key, and then retrieve messages from the queue.

The RabbitMQ management console can also be used to send messages to the exchange so that they can be consumed by a component. (The exchange used by SFM is named *sfm_exchange*.)

For more information on the RabbitMQ management console, see *RabbitMQ*.

Blocked ports

When running on a remote VM, some ports (e.g., 15672 used by the RabbitMQ management console) may be blocked. SSH port forwarding can help make those ports available.

Django logs

Django logs for SFM UI are written to the Apache logs. In the docker environment, the level of various loggers can be set from environment variables. For example, setting *SFM_APSCHEDULER_LOG* to *DEBUG* in the *docker-compose.yml* will turn on debug logging for the apscheduler logger. The logger for the SFM UI application is called ui and is controlled by the *SFM_UI_LOG* environment variable.

Apache logs

In the SFM UI container, Apache logs are sent to stdout/stderr which means they can be viewed with *docker-compose logs* or *docker logs <container name or id>*.

Initial data

The development and master docker images for SFM UI contain some initial data. This includes a user ("testuser", with password "password"). For the latest initial data, see *fixtures.json*. For more information on fixtures, see the Django docs.

Runserver

There are two flavors of the the development docker image for SFM UI. *gwul/sfm-ui:dev* runs SFM UI with Apache, just as it will in production. *gwul/sfm-ui:dev-runserver* runs SFM UI with runserver, which dynamically reloads changed Python code. To switch between them, change the *image* field in your *docker-compose.yml*.

Job schedule intervals

To assist with testing and development, a 5 minute interval can be added by setting *SFM_FIVE_MINUTE_SCHEDULE* to *True* in the *docker-compose.yml*.

1.5.3 Docker tips

Building vs. pulling

Containers are created from images. Images are either built locally or pre-built and pulled from Docker Hub. In both cases, images are created based on the docker build (i.e., the Dockerfile and other files in the same directory as the Dockerfile).

In a docker-compose.yml, pulled images will be identified by the *image* field, e.g., *image: gwul/sfm-ui:dev*. Built images will be identified by the *build* field, e.g., *build: app-dev*.

In general, you will want to use pulled images. These are automatically built when changes are made to the Github repos. You should periodically execute *docker-compose pull* to make sure you have the latest images.

You may want to build your own image if your development requires a change to the docker build (e.g., you modify fixtures.json).

Killing, removing, and building in development

Killing a container will cause the process in the container to be stopped. Running the container again will cause process to be re-started. Generally, you will kill and run a development container to get the process to be run with changes you've made to the code.

Removing a container will delete all of the container's data. During development, you will remove a container to make sure you are working with a clean container.

Building a container creates a new image based on the Dockerfile. For a development image, you only need to build when making changes to the docker build.

1.6 Docker

This page contains information about Docker that is useful for installation, administration, and development.

1.6.1 Installing Docker

Docker Engine and Docker Compose

On OS X:

- Install the Docker Toolbox.
- Be aware that Docker is not running natively on OS X, but rather in a VirtualBox VM.

On Ubuntu:

- If you have difficulties with the `apt` install, try the `pip` install.
- The docker group is automatically created. Adding your user to the docker group avoids having to use sudo to run docker commands. Note that depending on how users/groups are set up, you may need to manually need to add your user to the group in `/etc/group`.

1.6.2 Helpful commands

`docker-compose up -d` Bring up all of the containers specified in the docker-compose.yml file. If a container has not yet been pulled, it will be pulled. If a container has not yet been built it will be built. If a container has been stopped ("killed") it will be re-started. Otherwise, a new container will be created and started ("run").

`docker-compose pull` Pull the latest images for all of the containers specified in the docker-compose.yml file with the *image* field.

`docker-compose build` Build images for all of the containers specified in the docker-compose.yml file with the *build* field. Add `--no-cache` to re-build the entire image (which you might want to do if the image isn't building as expected).

`docker ps` List running containers. Add `-a` to also list stopped containers.

`docker-compose kill` Stop all containers.

`docker kill <container name>` Stop a single container.

`docker-compose rm -v --force` Delete the containers and volumes.

`docker rm -v <container name>` Delete a single container and volume.

`docker rm $(docker ps -a -q) -v` Delete all containers.

`docker-compose logs` List the logs from all containers.

`docker logs <container name>` List the log from a single container.

`docker-compose -f <docker-compose.yml filename> <command>` Use a different docker-compose.yml file instead of the default.

`docker exec -it <container name> /bin/bash` Shell into a container.

`docker rmi <image name>` Delete an image.

`docker rmi $(docker images -q)` Delete all images

`docker-compose scale <service name>=<number of instances>` Create multiple instances of a service.

1.6.3 Scaling up with Docker

To create multiple instances of a service, use docker-compose scale. This can be used to created multiple instances of a harvester when the queue for that harvester is too long.

To spread containers across multiple containers, use Docker Swarm.

Using compose in production provides some additional guidance.

1.7 Writing a harvester

1.7.1 Requirements

- Implement the Messaging Specification for harvesting social media content. This describes the messages that must be consumed and produced by a harvester.
- Write harvested social media to a WARC, following all relevant guidelines and best practices. The message for announcing the creation of a WARC is described in the Messaging Specification. The WARC file must be written to *<base path>/<harvest year>/<harvest month>/<harvest day>/<harvest hour>/*, e.g.,

/data/test_collection_set/2015/09/12/19/. (Base path is provided in the harvest start message.) Any filename may be used but it must end in *.warc* or *.warc.gz*. It is recommended that the filename include the harvest id (with file system unfriendly characters removed) and a timestamp of the harvest.

- Extract urls for related content from the harvested social media content, e.g., a photo included in a tweet. The message for publishing the list of urls is described in the Messaging Specification.
- Document the harvest types supported by the harvester. This should include the identifier of the type, the API methods called, the required parameters, the optional parameters, what is included in the summary, and what urls are extracted. See the Flickr Harvester as an example.
- The smoke tests must be able to prove that a harvester is up and running. At the very least, the smoke tests should check that the queues required by a harvester have been created. (See test_queues().)
- Be responsible for its own state, e.g., keeping track of the last tweet harvested from a user timeline. See sfmutils.state_store for re-usable approaches to storing state.
- Create all necessary exchanges, queues, and bindings for producing and consuming messages as described in Messaging.
- Provide master and production Docker images for the harvester on Docker Hub. The master image should have the *master* tag and contain the latest code from the master branch. (Setup an automated build to simplify updating the master image.) There must be a version specific production images, e.g., *1.3.0* for each release. For example, see the Flickr Harvester's dockerfiles and Docker Hub repo.

1.7.2 Suggestions

- See sfm-utils for re-usable harvester code. In particular, consider subclassing BaseHarvester.
- Create a development Docker image. The development Docker images links in the code outside of the container so that a developer can make changes to the running code. For example, see the Flickr harvester development image.
- Create a development *docker-compose.yml*. This should include the development Docker image and only the additional images that the harvester depends on, e.g., a Rabbit container. For example, see the Flickr harvester development docker-compose.yml.
- When possible, use existing API libraries.
- Consider write integration tests that test the harvester in an integration test environment. (That is, an environment that includes the other services that the harvester depends on.) For example, see the Flickr Harvester's integration tests.
- See the Twitter harvester unit tests for a pattern on configuring API keys in unit and integration tests.

1.7.3 Notes

- Harvesters can be written in any programming language.
- Changes to gwu-libraries/* repos require pull requests. Pull requests are welcome from non-GWU developers.

1.8 Messaging

1.8.1 RabbitMQ

RabbitMQ is used as a message broker.

The RabbitMQ managagement console is exposed at `http://<your docker host>:15672/`. The username is `sfm_user`. The password is the value of `RABBITMQ_DEFAULT_PASS` in `secrets.env`.

1.8.2 Publishers/consumers

- The hostname for RabbitMQ is `mq` and the port is 5672.
- It cannot be guaranteed that the RabbitMQ docker container will be up and ready when any other container is started. Before starting, wait for a connection to be available on port 5672 on `rabbit`. See appdeps.py for docker application dependency support.
- Publishers/consumers may not assume that the requisite exchanges/queues/bindings have previously been created. They must declare them as specified below.

1.8.3 Exchange

`sfm_exchange` is a durable topic exchange to be used for all messages. All publishers/consumers must declare it.:

```
#Declare sfm_exchange
from kombu import Connection

exchange = Exchange(name="sfm_exchange,
                    type="topic", durable=True)
exchange(channel).declare()
```

1.8.4 Queues

All queues must be declared durable.:

```
#Declare harvester queue
from kombu import Queue
queue = Queue(name="harvester",
              exchange=exchange,
              channel=channel,
              durable=True)
queue.declare()
queue.bind_to(exchange=exchange,
              routing_key="harvest.status.*.*")
```

1.9 Messaging Specification

1.9.1 Introduction

SFM is architected as a number of components that exchange messages via a messaging queue. To implement functionality, these components send and receive messages and perform certain actions. The purpose of this document is to describe this interaction between the components (called a "flow") and to specify the messages that they will exchange.

Note that as additional functionality is added to SFM, additional flows and messages will be added to this document.

1.9.2 General

- Messages may include extra information beyond what is specified below. Message consumers should ignore any extra information.
- RabbitMQ will be used for the messaging queue. See the Messaging docs for additional information. It is assumed in the flows below that components receive messages by connecting to appropriately defined queues and publish messages by submitting them to the appropriate exchange.

1.9.3 Harvesting social media content

Harvesting is the process of retrieving social media content from the APIs of social media services and writing to WARC files. It also includes extracting urls for other web resources from the social media so that they can be harvested by a web harvester. (For example, the link for an image may be extracted from a tweet.)

Background information

- A requester is an application that requests that a harvest be performed. A requester may also want to monitor the status of a harvest. In the current architecture, the SFM UI serves the role of requester.
- A stream harvest is a harvest that is intended to continue indefinitely until terminated. A harvest of a Twitter public stream is an example of a stream harvest. A stream harvest is different from a non-stream harvest in that a requester must both start and optionally stop a stream harvest. Following the naming conventions from Twitter, a harvest of a REST, non-streaming API will be referred to as a REST harvest.
- Depending on the implementation, a harvester may produce a single warc or multiple warcs. It is likely that in general stream harvests will result in multiple warcs, but REST harvest will result in a single warc.

Flow

The following is the flow for a harvester performing a REST harvest and creating a single warc:

1. Requester publishes a harvest start message.
2. Upon receiving the harvest message, a harvester:
   (a) Makes the appropriate api calls.
   (b) Extracts urls for web resources from the results.
   (c) Writes the api calls to a warc.
3. Upon completing the api harvest, the harvester:
   (a) Publishes a web harvest message containing the extracted urls.
   (b) Publishes a warc created message.
   (c) Publishes a harvest status message with the status of *completed success* or *completed failure*.

The following is the message flow for a harvester performing a stream harvest and creating multiple warcs:

1. Requester publishes a harvest start message.
2. Upon receiving the harvest message, a harvester:
   (a) Opens the api stream.
   (b) Extracts urls for web resources from the results.
   (c) Writes the stream results to a warc.

3. When rotating to a new warc, the harvester publishes a warc created message.
4. At intervals during the harvest, the harvester:
   (a) Publishes a web harvest message containing extracted urls.
   (b) Publishes a harvest status message with the status of *running*.
5. When ready to stop, the requester publishes a harvest stop message.
6. Upon receiving the harvest stop message, the harvester:
   (a) Closes the api stream.
   (b) Publishes a final web harvest message containing extracted urls.
   (c) Publishes a final warc created message.
   (d) Publishes a final harvest status message with the status of *completed success* or *completed failure*.

- Any harvester may send harvest status messages with the status of *running* before the final harvest status message. A harvester performing a stream harvest must send harvest status messages at regular intervals.
- A requester should not send harvest stop messages for a REST harvest. A harvester performing a REST harvest may ignore harvest stop messages.

Messages

Harvest start message

Harvest start messages specify for a harvester the details of a harvest. Example:

```
{
    "id": "sfmui:45",
    "type": "flickr_user",
    "path": "/sfm-data/collections/3989a5f99e41487aaef698680537c3f5/6980fac666c54322a2ebdbcb2a9510f5"
    "seeds": [
        {
            "id": "a36fe186fbfa47a89dbb0551e1f0f181",
            "token": "justin.littman",
            "uid": "131866249@N02"
        },
        {
            "id": "ab0a4d9369324901a890ec85f00194ac",
            "token": "library_of_congress"
        }
    ],
    "options": {
        "sizes": ["Thumbnail", "Large", "Original"]
    },
    "credentials": {
        "key": "abddfe6fb8bba36e8ef0278ec65dbbc8",
        "secret": "1642649c54cc3ebe"
    },
    "collection_set": {
        "id": "3989a5f99e41487aaef698680537c3f5"
    }
}
```

Another example:

```
{
    "id": "test:1",
    "type": "twitter_search",
    "path": "/sfm-data/collections/3989a5f99e41487aaef698680537c3f5/6980fac666c54322a2ebdbcb2a9510f5"
    "seeds": [
        {
            "id": "32786222ef374eb38f1c5d56321c99e8",
            "token": "gwu"
        },
        {
            "id": "0e789cddd0fb41b5950f569676702182",
            "token": "gelman"
        }
    ],
    "credentials": {
        "consumer_key": "EHde7ksBGgflbP5nUalEfhaeo",
        "consumer_secret": "ZtUpemtBkf2maqFiy52D5dihFPAiLebuMOmqN0jeQtXeAlen",
        "access_token": "481186914-c2yZjgbk13np0Z5MWEFQKSQNFBXd8T9r4k90YkJl",
        "access_token_secret": "jK9QOmn5Vbbmfg2ANT6KgfmKRqV8ThXVQ1G6qQg8BCejvp"
    },
    "collection_set": {
        "id": "3989a5f99e41487aaef698680537c3f5"
    }
}
```


- The routing key will be *harvest.start.<social media platform>.<type>*. For example, *harvest.start.flickr.flickr_photo*.
- *id*: A globally unique identifier for the harvest, assigned by the requester.
- *type*: Identifies the type of harvest, including the social media platform. The harvester can use this to map to the appropriate api calls.
- *seeds*: A list of seeds to harvest. Each seed is represented by a map containing *id*, *token* and (optionally) *uid*. Note that some harvest types may not have seeds.
- *options*: A name/value map containing additional options for the harvest. The contents of the map are specific to the type of harvest. (That is, the seeds for a flickr photo are going to be different than the seeds for a twitter user timeline.)
- *credentials*: All credentials that are necessary to access the social media platform. Credentials is a name/value map; the contents are specific to a social media platform.
- *path*: The base path for the collection.

Web resource harvest start message Harvesters will extract urls from the harvested social media content and publish a web resource harvest start message. This message is similar to other harvest start messages, with the differences noted below. Example:


```
{
    "id": "flickr:45",
    "parent_id": "sfmui:45",
    "type": "web",
    "path": "/sfm-data/collections/3989a5f99e41487aaef698680537c3f5/6980fac666c54322a2ebdbcb2a9510f5"
    "seeds": [
        {
            "id": "3724fd97e85345ee84f5175eee09748d",
            "token": "http://www.gwu.edu/"
        },
```

```
        {
            "id": "aba6033aafce4fbabd846026ca47f13e",
            "token": "http://library.gwu.edu/"
        }
    ],
    "collection_set": {
        "id": "3989a5f99e41487aaef698680537c3f5"
    }
}
```

- The routing key will be *harvest.start.web*.
- *parent_id*: The id of the harvest from which the urls were extracted.

Harvest stop message

Harvest stop messages tell a harvester perform a stream harvest to stop. Example:

```
{
    "id": "sfmui:45"
}
```

- The routing key will be *harvest.stop.<social media platform>.<type>*. For example, *harvest.stop.twitter.filter*.

Harvest status message

Harvest status messages allow a harvester to provide information on the harvests it performs. Example:

```
{
    "id": "sfmui:45"
    "status": "completed success",
    "date_started": "2015-07-28T11:17:36.640044",
    "date_ended": "2015-07-28T11:17:42.539470",
    "infos": []
    "warnings": [],
    "errors": [],
    "stats": {
        "2016-05-20": {
            "photos": 12,
        },
        "2016-05-21": {
            "photos": 19,
        },
    },
    "token_updates": {
        "a36fe186fbfa47a89dbb0551e1f0f181": "j.littman"
    },
    "uids": {
        "ab0a4d9369324901a890ec85f00194ac": "671366249@N03"
    },
    "warcs": {
        "count": 3
        "bytes": 345234242
    }
}
```

- The routing key will be *harvest.status.<social media platform>.<type>*. For example, *harvest.status.flickr.flickr_photo*.
- *status*: Valid values are *completed success*, *completed failure*, or *running*.
- *infos*, *warnings*, and *errors*: Lists of messages. A message should be an object (i.e., dictionary) containing a *code* and *message* entry. Codes should be consistent to allow message consumers to identify types of messages.
- *stats*: A count of items that are harvested by date. Items should be a human-understandable labels (plural and lower-cased). Stats is optional for in progress statuses, but required for final statuses.
- *token_updates*: A map of uids to tokens for which a token change was detected while harvesting. For example, for Twitter a token update would be provided whenever a user's screen name changes.
- *uids*: A map of tokens to uids for which a uid was identified while harvesting at not provided in the harvest start message. For example, for Flickr a uid would be provided containing the NSID for a username.
- *warcs*.'count': The total number of WARCs created during this harvest.
- *warcs*.'bytes': The total number of bytes of the WARCs created during this harvest.

Warc created message

Warc created message allow a harvester to provide information on the warcs that are created during a harvest. Example:

```
{
    "warc": {
        "path": "/sfm-data/collections/3989a5f99e41487aaef698680537c3f5/6980fac666c54322a2ebdbcb2a951
        "sha1": "7512e1c227c29332172118f0b79b2ca75cbe8979",
        "bytes": 26146,
        "id": "aba6033aafce4fbabd846026ca47f13e",
        "date_created": "2015-07-28T11:17:36.640178"
    },
    "collection_set": {
        "id": "3989a5f99e41487aaef698680537c3f5"
    },
    "harvest": {
        "id": "98ddaa6e8c1f4b44aaca95bc46d3d6ac",
        "type": "flickr_user"
    }
}
```

- The routing key will be *warc_created*.
- Each warc created message will be for a single warc.

1.9.4 Exporting social media content

Exporting is the process of extracting social media content from WARCs and writing to export files. The exported content may be a subset or derivate of the original content. A number of different export formats will be supported.

Background information

- A requester is an application that requests that an export be performed. A requester may also want to monitor the status of an export. In the current architecture, the SFM UI serves the role of requester.
- Depending on the nature of the export, a single or multiple files may be produced.

Flow

The following is the flow for an export:

1. Requester publishes an export start message.
2. Upon receiving the export start message, an exporter:
   (a) Makes calls to the SFM REST API to determine the WARC files from which to export.
   (b) Limits the content is specified by the export start message.
   (c) Writes to export files.
3. Upon completing the export, the exporter publishes an export status message with the status of *completed success* or *completed failure*.

Export start message

Export start messages specify the requests for an export. Example:

```
{
    "id": "f3ddcbfc5d6b43139d04d680d278852e",
    "type": "flickr_user",
    "collection": {
        "id": "005b131f5f854402afa2b08a4b7ba960"
    },
    "path": "/sfm-data/exports/45",
    "format": "csv",
    "dedupe": true,
    "item_date_start": "2015-07-28T11:17:36.640178",
    "item_date_end": "2016-07-28T11:17:36.640178",
    "harvest_date_start": "2015-07-28T11:17:36.640178",
    "harvest_date_end": "2016-07-28T11:17:36.640178"
}
```

Another example:

```
{
    "id": "f3ddcbfc5d6b43139d04d680d278852e",
    "type": "flickr_user",
    "seeds": [
        {
            "id": "48722ac6154241f592fd74da775b7ab7",
            "uid": "23972344@N05"
        },
        {
            "id": "3ce76759a3ee40b894562a35359dfa54",
            "uid": "85779209@N08"
        }
    ],
    "path": "/sfm-data/exports/45",
    "format": "json"
}
```

- The routing key will be *export.start.<social media platform>.<type>*. For example, *export.start.flickr.flickr_user*.
- *id*: A globally unique identifier for the harvest, assigned by the requester.

- *type*: Identifies the type of export, including the social media platform. The export can use this to map to the appropriate export procedure.
- *seeds*: A list of seeds to export. Each seed is represented by a map containing *id* and *uid*.
- *collection*: A map containing the *id* of the collection to export.
- Each export start message must have a *seeds* or *collection* but not both.
- *path*: A directory into which the export files should be placed. The directory may not exist.
- *format*: A code for the format of the export. (Available formats may change.)
- *dedupe*: If true, duplicate social media content should be removed.
- *item_date_start* and *item_date_end*: The date of social media content should be within this range.
- *harvest_date_start* and *harvest_date_end*: The harvest date of social media content should be within this range.

Export status message

Export status messages allow an exporter to provide information on the exports it performs. Example:

```
{
    "id": "f3ddcbfc5d6b43139d04d680d278852e"
    "status": "completed success",
    "date_started": "2015-07-28T11:17:36.640044",
    "date_ended": "2015-07-28T11:17:42.539470",
    "infos": []
    "warnings": [],
    "errors": [],
}
```

- The routing key will be *export.status.<social media platform>.<type>*. For example, *export.status.flickr.flickr_user*.
- *status*: Valid values are *completed success* or *completed failure*.
- *infos*, *warnings*, and *errors*: Lists of messages. A message should be an object (i.e., dictionary) containing a *code* and *message* entry. Codes should be consistent to allow message consumers to identify types of messages.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

2.1 Funding history

- Development of this project has been supported by a grant (#NARDI-14-50017-14) from the National Historical Publications & Records Commission to George Washington University Libraries from 2014-2017.
- Development of the Sina Weibo harvester is supported by a grant from the Council on East Asian Libraries.
- **Prior development of SFM under the previous repository** was supported by a grant (#LG-46-13-0257-13) from the Institute of Museum and Library Services to George Washington University Libraries from 2013-2014.