Webservices

Zach Pearson
RBVI Developer Meeting
27 January 2022

What are webservices?

- We run computations too expensive for user machines to run
- Traditionally, webservices communicate with XML over HTTP
    - SOAP is used to transmit messages
    - Web Services Description Language is used to define a service
- Advantages of SOAP:
    - Clear pattern with less work to do bootstrapping your API
- Drawbacks:
    - Lots of XML overhead for messages
    - Inflexible formatting
    - **Main drawback:** A *cultural* shift away from SOAP is contributing to bit rot in supporting infrastructure

REST

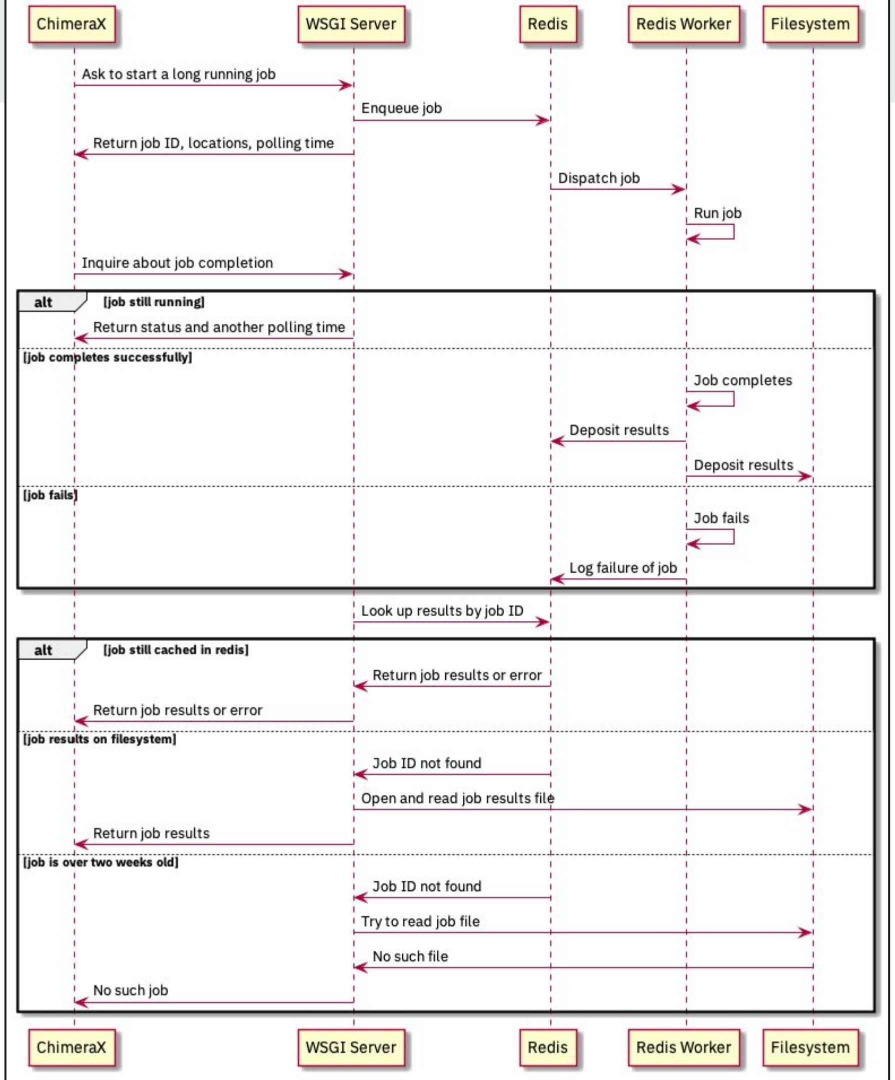
- REpresentational State Transfer
  - Underlying message format does not matter (can be XML or JSON)
    - Most people prefer JSON
- REST is not a standard but a prescription for how to use other standards: URIs, HTTP, MIME types
- Use HTTP as it was originally envisioned: HTTP verbs act on concrete resources
- An API is RESTful if it conforms to some or all of the following:
  - URIs identify resources, actions on resources should be constrained by HTTP
  - Client keeps session state: requests should contain all information necessary to interpret and complete the request
  - Representation decoupled from underlying resource (can return HTML, XML, JSON if desired)
  - Representations tell clients how they can be used
- Anything possible with SOAP is possible with REST

How We Conform to REST

- Base URL:
  - `http://webservices.rbvi.ucsf.edu/cxservices/api/v1`
- Resources:
  - `/chimerax/services`
    - `GET` to list all services
  - `/chimerax/services/{service}`
    - `POST` to a service to submit a job
  - `/chimerax/services/{service}/status`
    - `GET` to check whether a service is online
  - `/chimerax/jobs/{job_id}`
    - `DELETE` to cancel jobs
  - `/chimerax/jobs/{job_id}/{status,results}`
    - `GET` the status or results of a job
  - `/chimerax/updates`
    - `GET` to see if there's a new version of ChimeraX
- The pattern:
  - Every URI contains, and ends with, nouns, and the verbs are all HTTP verbs. There's no /update /delete etc.

Architecture

Architecture


ChimeraX
WSGI Server
Redis
Redis Worker
Filesystem
Ask to start a long running job
Enqueue job
Return job ID, locations, polling time
Dispatch job
Run job
Inquire about job completion
alt
[job still running]
Return status and another polling time
[job completes successfully]
Job completes
Deposit results
Deposit results
[job fails]
Job fails
Log failure of job
Look up results by job ID
alt
[job still cached in redis]
Return job results or error
Return job results or error
[job results on filesystem]
Job ID not found
Open and read job results file
Return job results
[job is over two weeks old]
Job ID not found
Try to read job file
No such file
No such job
ChimeraX
WSGI Server
Redis
Redis Worker
Filesystem

How We Load Webservices on Plato

Server definition

- `/usr/local/www/webservices-conf.d/VirtualHosts/webservices.conf`
  - `WSGIDaemonProcess` gives us our own pgroup
  - `WSGIScriptAlias` gives us a base url, like `/cxservices/api/v1`
    - relative to `webservices.rbvi.ucsf.edu` domain
    - Second “argument” points to a WSGI file that will implement any sub-URLs

Python interface

- Webservices lives at `/usr/local/www/webservices/wsgi-scripts/cx_v1.wsgi`
  - Imports the local copy of the `cxwebservices` repo
  - Calls `app.py:create_app()`

After that, we’re live!

Local Development

How to develop webservices locally

- Get the code and prepare your environment
  - Install `redis` on your hardware or source a `redis` docker container
    - Map port 6379 to the docker container's port 6379
  - Clone `git@github.com/RBVI/cxwebservices` to your preferred destination
  - `pip install pipenv` using the system Python
  - Navigate to the project directory, then `pipenv install`
  - Activate the environment with `pipenv shell`
- Source necessary 3rd party binaries from Plato, the internet, or local compilation
- Start a local server for development
  - `cd` to the directory just above `cxwebservices`
  - `waitress-serve --listen=localhost:8000 --call cxwebservices.app:create_app`
  - In a new terminal: `rq worker --url=redis://localhost:6379`

Regenerating the ChimeraX client

- Install openjdk8 (there is a Brewfile for macOS users to do this)
- Edit line 3 of the makefile to say ‘test’ or ‘production’
- Edit cxservices.yml.in
- `make client`
- `make upload_new_version`
- Can also copy the wheel to `ChimeraX/prereqs/cxservices`

The Job API

Job API

- Every module in the `task_runners` folder defines a webservice we offer
- The name of the file has meaning to the API
  - Filenames are taken as service names
  - 1:1 relationship between filename and `{base_url}/chimerax/services/{service_name}`
- Two required methods
  - `validate_params(*, ...) -> None`
  - `run_job(job_id, *, ...) -> Optional[Any]`
- Two optional methods
  - `setup_job(...) -> None` (EXPERIMENTAL)
  - `check_status() -> bool`
- One optional class
  - `class PollTimer()`

The Service Registry Hack

```python
# JobManager.py
avail_mods_bef = set(globals())
from ..task_runners import *
avail_mods_aft = set(globals())
avail_mods_set = avail_mods_bef ^ avail_mods_aft
aval_mods_set.remove('avail_mods_bef')
available_modules = {}
for value in avail_mods_set:
    available_modules[value] = globals()[value]
del avail_mods_bef, avail_mods_aft, avail_mods_set
```

`setup_job(job_id, **kwargs) -> None`

- Arbitrary data can be `POST`ed to a service: parameters and their values, files...
- `setup_job()` should handle the binary data needed to perform a job

When to use it:

- BLAST and BLAT don't need it, because they take in literal parameters and write a single results file
- Modeller will need it, because we need to make a directory, write some files inside of it, run Modeller, etc.

What's left to make it a reality:

- JobManager should parse multipart data and create a list of literal arguments and files, then pass them both to setup_job to handle the rest

`validate_params(*, ...) -> None`

- JobManager will call `validate_params(**params)`
  - Same as the JavaScript spread operator, expands the params dict
- Either `validate_params(*, key1, ...)` or `validate_params(**kwargs)` works
  - `(*, key1, ...)` enforces correct name and number of params at call time
  - `(**kwargs)` gives you more flexibility but requires a more verbose definition (`if kwargs.get(...)`)
- `raise ValueError` if unexpected/out of range values received

`run_job(job_id, *, ...) -> Optional[Any]`

- JobManager calls `self.job_queue.enqueue(job_id, **params, job_id=job_id)`
- Define the entire sequence of actions required to run a job as if it was a local python script
- Any action that the `apache` account can do is possible
- Any returned values are stored in redis
- May not need to pass in job_id in the future; see https://python-rq.org/docs/jobs/#accessing-the-current-job-from-within-the-job-function
    - If leveraged, could store arbitrary information in redis from within the job
    - Careful: run_job will be the only function in your service definition file that has the job context

`check_status() -> bool`

- Return information about the availability of a certain resource
- Most resources we control will return UP
- Services out of our control such as AlphaFold
- In the future we would like to cache the results for some amount of time
    - Reduce server load and requests to external resources

`class PollTimer()`

- `cxservices/utils.py` includes a generic poll timer, so this class is optional
- If you have an especially long running job and want to change the way ChimeraX polls the backend for results, implement the PollTimer class in your task runner.
- Required:
  - `__init__(self)`
  - `__iter__(self)`
  - `__next__(self)`
- The default sequence is 1, 2, 3, ..., 10

Adding a Webservice

Adding a Webservice

1. Create a file in `task_runners/` with the name of the service
2. Define at least `run_job(job_id, *, ...)` and `validate_params(*, ...)`
3. Edit `task_runners/__init__.py`
   a. `from .{your_module} import *`
   b. `__all__ = [“blast”, ..., “your_module”]`

Things should just work.

Webservices Are Not Routes

- Every webservice conforms to the `/chimerax/services/{service}/{resource}` pattern
- `cxservices/service_cxnewer.py` is an example of a file that defines a route but is not a webservice
    - Adds the `/chimerax/updates` and `/chimerax/newer` routes
- Primary difference is that cxnewer is fast enough to run synchronously
- To add a route
    - Create a file
    - Define `add_routes(api) -> None`
    - Call `api.add_route(route: str, handling_class, suffix: Optional[str])` to add your route
    - Import it in `cxservices/__init__.py`
    - Add a call to `module.add_routes(api)` in the `__init__`-level `add_routes`

Testing / Continuous Integration

Why test?

- Automated testing can greatly increase the velocity of development
  - Passing local tests reduces the estimated workload of remote testing to integration tests
- We can enforce
  - The contract the API specifies
  - (a set of sensible) flake8 rules
- We can collect
  - Code coverage statistics – increase the confidence of our testing suite by increasing lines audited
- A proven system passing automated tests can in principle be automatically staged for deployment

What to test

- Tests of individual functions should enforce an API's contract, not its implementation details
  - For example, existing tests for `cxservices/utils.py`:
    - Assert that we will return a 64 character, capital alphanumeric job ID
    - Assert that the default poll intervals are 1, 2, 3, …, 10
- Tests of task runners should be integration tests
  - For example, existing tests for `task_runners/blast.py:`
    - Enforce preservation of 1.3 routes
    - Ensure systems work together in principle

Testing a Service

1. Create a file in `tests/` that follows the `test_{service}.py` pattern
2. Define at least one `test_foo(...)` **method**
3. Run all tests: `coverage run -m pytest`
4. Run a specific test: `coverage run -m pytest path/to/file.py`

Future Development

AlphaFold User Priority Queue

- Webservices currently has *one* queue

<u>But</u>: we can have an arbitrary number of queues with different priorities

- If we send the client UUID with every request, we can store the UUID in a database
  - Every time we get a request to `/chimerax/services/{service}` increment a counter associated with the ID
  - Someday let's argue about whether MySQL or MongoDB is the better choice
- Set sensible thresholds for deprioritizing requests from UUIDs based on how many jobs they've submitted
- Reset the count every (day, week, month)

Update Bundles from Webservices

- On the server:
  - Create an authenticated route to upload prerequisites to prereqs
  - Create an authenticated route to upload bundles to the toolshed
- On the client, starting with webservices itself:
  - Create a programmatic interface to the toolshed
  - Define a minimum Python version required for each bundle based on features
    - Example: `match...case` requires Python 3.10
  - Expand the `/chimerax/updates` route to `/chimerax/updates/{bundle,chimerax}`
  - Use the toolshed machinery to download and install updates automatically, for as long as those updates and ChimeraX-Core are compatible with the built-in Python environment
  - Start with pure Python bundles and expand to binary bundles in the future

Add Opal-like Introspection

- Authenticated route to show a page that gives information about the state of the system
  - Number of running jobs
  - 24 hour average
  - Availability?

Automated Staging to Webservices-Test

- The environment on Plato is actually modelable as a single machine
- The webservices infrastructure, if it fails over, fails over together
  - Think about restarting a race rather than pausing it, moving it, and resuming it
- Because of this, we can rely on integration tests in GitHub actions to “prove” readiness to stage code changes to Webservices-Test
- GitHub Actions can be set to send a POST request to an arbitrary URL with an auth token we set
- We could define a route which will automatically update the server
  - The in-memory server is not affected by code changes
- This route could then `apache_graceful webservices-test`
- Requires that the apache account can run apache_graceful
- Hand integration testing in local ChimeraX builds can then begin

The Only Limit is Dev Time