Web service Documentation

Release SVN

Sali Lab

Oct 08, 2021

Contents

The lab web framework provides a simple set of classes and modules to simplify the process of deploying a web service (a web-based front end to jobs that run on the cluster).

Developers eager to get started may want to jump straight to *Quick start*.

CHAPTER 1

Contents

1.1 Introduction

The Sali lab web framework aims to provide a simple set of classes and modules to simplify the process of deploying a web service. It is designed for typical Sali lab web applications - that is to say, jobs that are submitted from a web interface but that then run on one or more cluster nodes for a possibly long period of time.

Developers eager to get started may want to jump straight to *Quick start*.

The framework is split up into four distinct parts:

- The *frontend* consists of HTML pages and Python scripts that provide a web interface to interact with an end user. It also provides an XML-based interface to allow automated submission of jobs. It handles the uploading of input files, the initial submission of jobs, displaying a queue of all jobs in the system, and showing the results of completed jobs. It can also potentially handle user logins.
- The *backend* is a set of Python classes that manages jobs after the initial submission. This typically runs as a daemon on our *modbase* machine, picking up submitted jobs from the frontend, submitting jobs to the cluster and gathering results, and doing any necessary pre- or post-processing.
- The *build system* is a set of extensions to SCons that simplifies the procedure of deploying a a web service (initial setup, and installing everything in the correct locations and with the right permissions).
- *Monitoring* is typically set up by a sysadmin, and ensures that the web service, once set up and made available to end users, is correctly functioning.

The web framework stores all persistent data about the jobs in a MySQL database, and configuration for the service is stored in a set of *configuration files* that are shared between all parts of the framework.

The framework aims to be:

- *Comprehensive*: can handle simple services like ModLoop and complex services like CM-MR.
- *Robust*: full error checking is done at each step of the process; job failures are caught and reported to the server admin.
- *Extensible*: a service can be set up without user authentication, but this can be simply added in later. The backend Python classes provide multiple hooks to allow each step of the job to be modified.

- *Secure*: by default, file permissions should be set up sensibly. There should be no need for world-writeable directories or other hacks to allow files to be passed from the web server to the cluster. The backend runs as a different user from the web server, so it is unlikely that a bug in the frontend can be used to break into the cluster.
- *Efficient*: it is straightforward to configure the system so that long-term data from jobs are not left on expensive disks such as NetApp, but are instead moved to park disks or deleted.
- *Easy to use*: given a simple workflow, it should be easy to deploy a simple web server that implements it.

1.2 Backend

The backend is a set of Python classes that manages jobs after the initial submission. This typically runs as a daemon on our 'modbase' machine, picking up submitted jobs from the frontend, submitting jobs to the cluster and gathering results, and doing any necessary pre- or post-processing.

The backend is implemented by the *saliweb.backend* Python module.

1.2.1 Classes

Job

The *Job* class represents a single job known to the backend. This could have just been submitted by the frontend, it could be running on the cluster, it could have finished and its results placed on long-term storage, or the results from a very old job could have been deleted (only the job metadata remains). Each job corresponds to a single row in the jobs database table and a directory on disk.

For any web service, the *Job* class must first be subclassed and then one or more of its methods implemented to actually do the work of running jobs. For example, the *Job.run()* method will be called by the backend when the job starts; it is expected to start the job running on the cluster, typically by using a *WyntonSGERunner* object. There are similar methods that can be used to do extra processing before the job starts (*Job.preprocess()*), after it finishes running on the cluster (*Job.postprocess()* and *Job.finalize()*) or when the job is moved to long-term storage (*Job.archive()*), for example. Each method is run in the directory containing all of the job's data (i.e. any files uploaded by the end user when the job was submitted, plus any output files after the job has run). (The exception is *Job.expire()*, which is called after the job directory has been deleted.) If any of these methods raises an exception, it is caught by the backend; the job is put into a failed state and the server admin is notified. Thus, exceptions should be used only to indicate a technical error in the web service, not something wrong with the user's input (in the latter case, the job output should simply indicate what the problem is).

Note: Each of these methods is automatically run by the backend at the correct time; they should not be run manually by any method in the subclass. For example, to run a new job, call *Job.reschedule_run()*, not the *Job.run()* method directly.

As mentioned above, a *WyntonSGERunner* class is provided that takes care of the details of running a script on the cluster and checking if it has completed. Typically the script run here should use the local /scratch disk on the cluster nodes if possible - this is not implemented automatically by the framework, since the best usage of local and network disks is specific to a given web service.

Database

Each job's metadata is stored in a database; the *Database* class manages this database and creates *Job* objects automatically when requested to by other classes. The base *Database* class interfaces with a MySQL database on

the 'modbase' machine and manages database tables containing fields used by all web services. It can be subclassed to add additional fields (for example, to store some extra job metadata in the database rather than within the job's directory) or (potentially) to use a different database engine.

Config

The `Config` class parses the configuration file for the web service and stores all of the configuration information. It can be subclassed if desired to read extra service-specific information from the configuration file, usually by extending the `Config.populate()` method.

WebService

The `WebService` class provides high-level backend functionality. The most commonly-used method is `WebService.do_all_processing()`, which simply runs in an endless loop, submitting new jobs to the cluster, collecting the results of finished jobs, and archiving old completed jobs. It is rarely necessary to subclass.

1.2.2 Job states

A single job in the system is represented by a row in the database table and a single directory that contains the job inputs and/or outputs. Each job can be in one of a set of distinct states, described below. In normal operation a job will move from the first state in the list below to the last.

- The **INCOMING** state is for jobs that have just been submitted to the system by the frontend, but not yet picked up by the frontend.
- Jobs move into the **PREPROCESSING** state when they are picked up by the frontend. At this point the `Job.preprocess()` method is called, which can be overridden to do any necessary preprocessing. Note that this method runs on the server machine ('modbase') and serially (for only a single job at a time), so it should not run any calculations that take more than a few seconds.
- Next, jobs usually move to the **RUNNING** state. At this point the `Job.run()` method is called, which typically will submit an SGE job to do the bulk of the processing.
- When the SGE job finishes, the job moves to the **POSTPROCESSING** state and the `Job.postprocess()` method is called. Like preprocessing, this runs serially on the server machine and so should not be computationally expensive.
- It may be decided in postprocessing that further runs are required, in which case the job moves back to the **RUNNING** state for another cycle. Otherwise, it moves to the **FINALIZING** state and `Job.finalize()` is called. This is the same as **POSTPROCESSING** except that is only called on the last cycle (no further runs can be started here).
- Next the job moves to the **COMPLETED** state and the `Job.complete()` method is called. If the user provided an email address to the frontend, they are emailed at this point to let them know job results are now available.
- After a defined period of time, the job moves to the **ARCHIVED** state and the `Job.archive()` method is called. At this point the job results are still present on disk, but are no longer accessible to the end user and may be moved to long-term storage.
- After another defined period of time, the job moves to the **EXPIRED** state, the job directory is deleted, and the `Job.expire()` method is called. At this point, only the job metadata in the database remains.

If a problem is encountered at any point (usually a Python exception) the job is moved to the **FAILED** state. At this point the server admin is emailed and is expected to fix the problem (usually a bug in the web service, or a system problem such as a broken or full hard disk).

Note also the *Job.preprocess()* method can, if desired, signal to the framework that running a full SGE job is unnecessary (by calling the *Job.skip_run()* method). In this case, the **RUNNING** and **POSTPROCESSING** steps are skipped and the job moves directly from **PREPROCESSING** to **COMPLETED**. Similarly, the *Job.postprocess()* method can request that the framework runs a new job (by calling the *Job.reschedule_run()* method). In this case, the job moves from **POSTPROCESSING** back to **RUNNING**.

Each job state (with the exception of **EXPIRED**) can be given a directory in the service's configuration file. Job data are automatically moved between directories when the state changes. For example, the **INCOMING** directory generally needs to reside on a local disk, and have special permissions so that the frontend can create files within it. The **RUNNING** directory usually needs to be accessible by the cluster, so it needs to be on the NetApp disk. The **ARCHIVED** directory may live on long-term storage, such as a park disk.

1.2.3 Examples

Simple job

The example below demonstrates a simple *Job* subclass that, given a set of PDB files from the frontend, runs an SGE job on the cluster that extracts all of the HETATM records from each PDB. This is done by overriding the *Job.run()* method to pass a set of shell script commands to an *WyntonSGERunner* instance; this instance is then returned to the backend. The backend will then keep track of the SGE job, and notice when it finishes.

The subclass also overrides the *Job.archive()* method, so that when the job results are moved from short-term to long-term storage, all of the PDB files are compressed with gzip to save space.

```
import saliweb.backend
import glob
import gzip
import os

class Job(saliweb.backend.Job):
    runnercls = saliweb.backend.WyntonSGERunner

    def run(self):
        script = """
for f in *.pdb; do
  grep '^HETATM' $f > $f.het
done
"""
        r = self.runnercls(script)
        r.set_options('-l diva1=1G')
        return r

    def archive(self):
        for f in glob.glob('*.pdb'):
            fin = open(f, 'rb')
            fout = gzip.open(f + '.gz', 'wb')
            fout.writelines(fin)
            fin.close()
            fout.close()
            os.unlink(f)
```

Custom database class

The *Database* class can be customized by adding additional fields to the database table. This is useful if you need to pass small amounts of job metadata between the frontend and backend, or between different stages of the job, and

the metadata are useful to keep after the job has finished.

Note: In many cases, it makes more sense to store job data as files in the job directory itself. For example, it is probably easier to store a PDB file as a real file rather than trying to insert the contents into the database table!

This example adds a new integer field *number_of_pdbs* to the database. The field can then be accessed (read or write) from within the *Job* object by referencing *self._metadata['number_of_pdbs']*. The *_metadata* attribute stores all of the job metadata in a Python dictionary-like object; it is essentially a dump of the database row corresponding to the job.

```
import saliweb.backend
import glob

class Database(saliweb.backend.Database):
    def __init__(self, jobcls):
        saliweb.backend.Database.__init__(self, jobcls)
        self.add_field(saliweb.backend.MySQLField('number_of_pdbs', 'INTEGER'))


class Job(saliweb.backend.Job):
    runnercls = saliweb.backend.WyntonSGERunner

    def preprocess(self):
        pdbs = glob.glob("*.pdb")
        self._metadata['number_of_pdbs'] = len(pdbs)

    def run(self):
        script = """
for f in *.pdb; do
  grep '^HETATM' $f > $f.het
done
"""
        r = self.runnercls(script)
        r.set_options('-l diva1=1G')
        return r
```

1.2.4 Logging

It is often useful for debugging purposes to log progress of a job. While the job is running on the cluster, the only way to do this is to write output into a log file. For other steps in the processing, however, the standard Python logging module is utilized. Each job method (such as *Job.run()*, *Job.preprocess()*) with the exception of *Job.expire()* can use the *Job.logger* object to write out log messages. It is a standard Python Logger object, so supports the regular methods of a Logger, such as `warning()` and `critical()` to write log messages, and `setLevel()` to set the threshold for log output.

By default, anything logged that exceeds the threshold will be written to a file called 'framework.log' in the job's directory. The file will only be created when the first log message is printed. This behavior can be modified if desired by overriding the *Job.get_log_handler()* method.

```
import saliweb.backend
import logging

class Job(saliweb.backend.Job):
    runnercls = saliweb.backend.WyntonSGERunner
```

(continues on next page)

(continued from previous page)

```
    def run(self):
        # Uncomment to get all logging output
        # self.logger.setLevel(logging.DEBUG)
        self.logger.info('Starting run method')
        script = """
for f in *.pdb; do
  grep '^HETATM' $f > $f.het
done
"""
        r = self.runnercls(script)
        self.logger.warning('Setting SGE options to diva1=1G')
        r.set_options('-l diva1=1G')
        self.logger.info('Ending run method')
        return r
```

1.2.5 Testing

The best way to test the backend is as part of the entire web service (see *Testing*).

However, the backend can be tested directly without invoking the frontend, by manually modifying the MySQL database. Note, however, that the interface between the backend and frontend, as well as the details of the MySQL tables, are not guaranteed to be stable (future iterations of the framework may change some of the details for performance or additional features), so this method could fail in future.

To manually submit a job:

1. Decide on a job name. This must be unique. Create a directory with the same name, as the backend user, under the web service's incoming directory (as specified in the configuration file).
2. Put all necessary input files into this directory.
3. Connect to the MySQL server using the *mysql* client on *modbase*, and the username and password from the web service's configuration file. Either the backend or frontend user can be used; the frontend user can only submit jobs and so is recommended, while the backend user can also delete or modify jobs, which is dangerous as it may break the service. For example, `mysql -u modfoo_frontend -p -D modfoo`.
4. To actually submit a job use something like:

```
INSERT INTO jobs (name,passwd,user,contact_email,
                  directory,url,submit_time)
                 VALUES (a,b,c,d..., UTC_TIMESTAMP());
```

   a,b,c,d are the values for the columns, described below:

- 'name' is the name of the job, from above.
- 'passwd' is used by the frontend to protect job results. Any alphanumeric string can be used here.
- 'user' is the user that submitted the job. NULL can be used here.
- 'contact_email' is the email address that the backend will notify when the job completes, or NULL for no email notification.
- 'directory' is the filesystem directory containing the job inputs, which must match that created above.

- 'url' is a web link that the backend will include in the email it sends out, telling the user where the results can be downloaded. A dummy value can be used here, since the frontend usually handles this.
- 'submit_time' is the time (UTC) when the job was submitted. Usually, the MySQL function UTC_TIMESTAMP() is used here to put in the current time.

5. The job will only be run if the backend is running (use the *bin/service.py* script as the backend user in the installation directory). The backend polls periodically for new jobs. Alternatively, *service.py* can be used to restart the backend, to force it to check immediately.

1.3 Configuration file

The configuration file is used to store information used by the *backend*, the *frontend*, and the build system. It is a fairly standard 'INI' file, containing section titles in square brackets (e.g. [general]) and key-value pairs within the sections, either of the form 'foo: bar' or 'foo = bar'.

The example file below defines the configuration for a fictional 'ModFoo' service:

```
[general]
admin_email: modfoo-admin@salilab.org
socket: /modbase1/home/modfoo/modfoo.socket
service_name: ModFoo
urltop: http://modbase.compbio.ucsf.edu/modfoo
google_ua: UA-44577804-1

[backend]
user: modfoo
state_file: /modbase1/home/modfoo/modfoo.state
check_minutes: 10

[limits]
running: 5

[database]
db: modfoo
backend_config: backend.conf
frontend_config: frontend.conf

[directories]
install: /modbase1/home/modfoo/
incoming: /modbase1/home/modfoo/incoming/
preprocessing: /wynton/home/sali/modfoo/running/
completed: /modbase1/home/modfoo/completed/
failed: /modbase1/home/modfoo/failed/

[oldjobs]
archive: 7d
expire: 30d
```

Each section in the configuration file is described below.

1.3.1 general

admin_email The email address of the administrator of this web service. This is used to notify the admin if a job fails with a technical error or the entire web service encounters an unrecoverable error and cannot continue.

socket The full path to a socket file that is used for the frontend to send messages to the backend.

service_name The name of the service (human-readable). This is used in emails to the owners of jobs and the server admin, and as the title of web pages.

urltop The URL under which the service's web pages live. This is used to construct URLs containing job results, for example.

google_ua The Google Analytics UA for tracking traffic. Please ask a sysadmin to register the service. As default, the ModBase UA is used.

track_hostname If set to "True" then the hostname or IP address of each web service user is stored in the database.

github For web services that are open source and are hosted on GitHub, this should be filled in with the GitHub URL.

1.3.2 backend

user The system user that the backend runs as. For security, robustness and easier monitoring, each web service has its own system user (e.g. the ModLoop web service runs as the 'modloop' system user). Note that the system user is distinct from the MySQL users set up to access the database.

state_file The full path to a file that is used by the backend to store state between calls. In normal operation this is simply used as a lock file to ensure that only one copy of the backend is running at a time. After an unrecoverable error, this file continues information on the nature of the failure and must be manually removed by the admin before the backend will run again.

check_minutes Typically, when new jobs are submitted the backend is notified and they start running immediately; once jobs are started the backend waits for them to finish and collects the results as soon as this happens. However, if a job is submitted from the frontend while the backend is not running, or the backend is restarted or the machine it is on is rebooted while jobs are running, the backend must fall back to a less efficient polling method to look for newly submitted or completed jobs. 'check_minutes' is the time, in minutes, to wait between checks for these jobs. An interval of 10 minutes is recommended. Note that archived and expired jobs are also checked for periodically, but this interval is fixed at 10% of the shorter of the archive and expiry times.

1.3.3 limits

running The maximum number of jobs that will run simultaneously. Defaults to 5.

concurrent_tasks The maximum number of tasks in each job that will run simultaneously. This is not enforced by the framework, but a service that runs a job on more than one machine can use this value to limit the parallelism. For example, services that run SGE array jobs can use this value to populate the *-tc* qsub parameter. Default is no limit.

1.3.4 database

db The name of the database in which the service's data are stored.

backend_config, frontend_config Filenames of additional INI files containing the MySQL username and password used by the backend and frontend to communicate with the database (in sections called [backend_db] and [frontend_db] respectively). (The frontend and backend should use different MySQL users, since they should have different access rights set up for the job tables.) If these filenames are not absolute paths, they are taken to be relative to the directory containing the main configuration file. The database authentication information has to stored in separate files so that file permissions can be set appropriately so that the frontend cannot read the backend configuration. An example backend.conf is shown below.

```
[backend_db]
user: modfoo_backend
passwd: Ra1Echoh4uim
```

1.3.5 directories

install The top-level directory in which the web service files are installed.

incoming, preprocessing, etc. Each *job state* except EXPIRED can be given a directory in which the job data are placed. Only the INCOMING and PREPROCESSING directories are required; others, if not specified, will default to the directory for the previous state (i.e. the RUNNING directory will default to that for PREPROCESSING, that for POSTPROCESSING will default to RUNNING, FINALIZING to POSTPROCESSING, COMPLETED to FINALIZING, and ARCHIVED to COMPLETED). If the FAILED directory is not given, it will default to the same as the COMPLETED directory.

1.3.6 oldjobs

This section controls what happens to old jobs after they have completed.

archive Completed job results, after this time, will no longer be available for the end user to download from the frontend. The time is either NEVER to indicate that job results are available forever, or a number with a single character suffix (h for hours, d for days, w for weeks, m for months or y for years). For example, '90d' will archive job results after 90 days.

expire Completed job results will be deleted from disk after this time. Times are specified in the same way as for *archive*. Note that the *archive* time cannot be longer than the *expire* time.

1.4 Frontend

The frontend is provided by Python code using the Flask microframework. This displays the web interface, allowing a user to upload their input files, start a job, display a list of all jobs in the system, and get back job results. This Python code uses utility classes and functions in the *saliweb.frontend* module, together with functionality provided by Flask.

1.4.1 Initialization

The first step is to create a Python module that creates the web application itself, using the *saliweb.frontend.make_application()* function (which in turn creates a Flask application and configures it). For a web service 'ModFoo' this should be done in the Python module `frontend/modfoo/__init__.py`. This is fairly standard boilerplate:

```
import flask
import saliweb.frontend


parameters = [...]
app = saliweb.frontend.make_application(__name__, parameters)
```

The 'parameters' object here is a list of parameters the job requires at submission time, and will be *described later*.

1.4.2 Web page layout

Each web page has a similar layout (header, footer, links, and so on). This is provided by a system-wide Jinja2 template called `saliweb/layout.html`, which can be seen at GitHub. This system-wide template should be overriden for each web service, by providing a file `frontend/modfoo/templates/layout.html` that 'extends' the template:

```
{% extends "saliweb/layout.html" %}

{%- block css %}
<link rel="stylesheet" type="text/css"
 href="{{ url_for("static", filename='modfoo.css') }}" />
{%- endblock %}

{% block navigation %}
{{ get_navigation_links(
       [(url_for("index"), "ModFoo Home"),
        (url_for("job"), "Current ModFoo queue"),
        (url_for("help"), "Help"),
        (url_for("contact"), "Contact")])
}}
{% endblock %}

{% block sidebar %}
<p><i>Version <a href="https://github.com/salilab/modfoo">{{ config.VERSION }}</a></i>
↪</p>
{% endblock %}

{% block footer %}
<p>
Please cite Bob et al., JMB 2008.
</p>
{% endblock %}
```

This example demonstrates a few Jinja2 and Flask features:

- Parts of the base template can be overriden using the `block` directive. Here, a custom stylesheet is added (by overriding the `css` block), links are added to all pages to the navigation bar at the top of the page (`navigation` block), and the sidebar and footer (at the left and bottom of the page, which are blank in the system-wide template) are filled in.
- Links to other parts of the web service can be provided using Flask's url_for function. This takes either the name of the Python function that renders the page (such as "index"; *see below*) or "static" to point to static files (such as stylesheets or images) in the web service's `html` subdirectory.
- A number of global variables are available which can be substituted in using Jinja2's `{{ }}` syntax, most notably `config` which stores web service configuration, such as the name and version in `config.SERVICE_NAME` and `config.VERSION` respectively.
- Some helper functions are available. Here the `get_navigation_links` function is used, which takes a list of (URL, description) pairs.

See the Jinja2 manual for more information on Jinja2 templates.

1.4.3 Displaying standard pages

The bulk of the functionality of the frontend is implemented by providing Python functions for each page using Flask routes.

For a typical web service, the index, submission, results and results file pages need to be implemented by providing `index`, `job`, `results`, and `results_file` functions, respectively. These pages will be considered in turn.

Note: Additional pages (such as page to download the software, or a help page) can be simply implemented by adding more Python functions with appropriate routes (and, if appropriate, adding links to these pages to `layout.html`). *See below*.

Index page

The index page is the first page seen when using the web service, and typically displays a form allowing the user to set parameters and upload input files. (In more complex web services this first form can lead to further forms for advanced options, etc.) The Python code for this is straightforward - it simply defines an `index` function which uses the Flask render_template function to render a Jinja2 template, and is then decorated using app.route to tell Flask to use this function to service the '/' URL:

```
@app.route('/')
def index():
    return flask.render_template('index.html')
```

The Jinja2 template in turn looks like:

```
{% extends "layout.html" %}

{% block body %}
<form method="post" action="{{ url_for("job") }}"
 enctype="multipart/form-data" name="modfooform">

  <p>Job name (optional)</p>
  <input type="text" name="job_name" />

  <p>Email address (optional)</p>
  <input type="text" name="email" value="{{ g.user.email }}" />

  <p>Upload PDB file</p>
  <input type="file" name="input_pdb" />
</form>
{% endblock %}
```

The template extends the previously-defined `layout.html` so will get the sidebar, footer, etc. defined there.

The form is set up to submit to the submission page (`url_for("job")`). It allows the user to upload a single PDB file, as well as pick an optional name for their job and an optional email address to be notified when the job completes.

Note that the email address is filled in using `g.user.email`. (`g` is used by Flask to store global data.) If the user is logged in to the webserver, `g.user` will be a *`LoggedInUser`* object, and their email address will be available for use in this fashion (otherwise, the user will simply have to input a suitable address if they want to be notified). Similarly, `g.user.modeller_key` provides the MODELLER license key of the logged-in user.

The names of the form parameters above (`job_name`, `input_pdb`) should also be described in the Python code `frontend/modloop/__init__.py` for automated use of the service (see *Automated use*), by passing suitable *`Parameter`* and/or *`FileParameter`* objects when the application is created. Note that `email` is omitted because automated usage typically does not use email notification:

```
import flask
import saliweb.frontend
```

(continues on next page)

(continued from previous page)

```
from saliweb.frontend import Parameter, FileParameter


parameters = [Parameter("job_name", "Job name", optional=True),
              FileParameter("input_pdb", "PDB file to be refined")]
app = saliweb.frontend.make_application(__name__, parameters)
```

Submission page

The submission page is called when the user has input all the information needed for the job. It is implemented by defining the `job` function which handles the `/job` URL. This serves double duty - an HTTP POST to this URL will submit a new job, while a GET will show all jobs in the system (the queue).

Showing the queue is handled by the *saliweb.frontend.render_queue_page()* function. Job submission should validate the provided information, then actually submit the job to the backend. If validation fails, it should throw an *InputValidationError* exception; this will be handled by the web framework as a message to the user asking them to fix the problem and resubmit. (If some kind of internal error occurs, such as a file write failure, in this or any other method, the user will see an error page and the server admin will be notified by email to fix the problem.)

To submit the job, first create a *saliweb.frontend.IncomingJob* object, which also creates a new directory for the job files. Put all necessary input files in that directory, for example using *IncomingJob.get_path()*, then actually run the job by calling *IncomingJob.submit()*.

Note: 'Input files' include PDB files, parameter files, etc. but *not* shell scripts, Python scripts, or executables. These should never be generated by the frontend, but instead should be generated by the backend. If these files are generated by the frontend, it is very easy for an unscrupulous end user to hack into the cluster by compromising the web service. The backend should always check inputs provided by the frontend (e.g. in the *preprocess()* method) for sanity before running anything.

Note: When taking files as input, you can simply write them into the job directory using their save method. But never trust the filename provided by the user! Ideally, save with a fixed generic name (e.g. `input.pdb`). If this is not possible, use the secure_filename function to get a safe version of the filename.

Finally, the submission page should inform the user of the results URL (*IncomingJob.results_url*), so that they can obtain the results when the job finishes. This uses the function *saliweb.frontend.render_submit_template()*, which will either display an HTML page (similarly to Flask's render_template, as before) or XML in the case of automated usage.

The example below reads in the PDB file provided by the user on the index page, checks to make sure it contains at least one ATOM record, then writes it into the job directory and finally submits the job:

```
@app.route('/job', methods=['GET', 'POST'])
def job():
    if flask.request.method == 'GET':
        return saliweb.frontend.render_queue_page()
    else:
        return submit_new_job()


def submit_new_job():
    # Get form parameters
```

(continues on next page)

(continued from previous page)

```
    input_pdb = flask.request.files.get('input_pdb')
    job_name = flask.request.form.get('job_name') or 'job'
    email = flask.request.form.get('email')

    # Validate input
    file_contents = input_pdb.readlines()
    atoms = 0
    for line in file_contents:
        if line.startswith('ATOM  '):
            atoms += 1
    if atoms == 0:
        raise saliweb.frontend.InputValidationError(
                  "PDB file contains no ATOM records!")

    # Create job directory, add input files, then submit the job
    job = saliweb.frontend.IncomingJob(job_name)

    with open(job.get_path('input.pdb'), 'w') as fh:
        fh.writelines(file_contents)

    job.submit(email)

    # Inform the user of the job name and results URL
    return saliweb.frontend.render_submit_template(
        'submit.html', email=email, job=job)
```

It uses the following template as `submit.html`, providing the `job` and `email` variables, to notify the user:

```
{% extends "layout.html" %}

{% block body %}
<p>Your job {{ job.name }} has been submitted.</p>

<p>Results will be found at <a href="{{ job.results_url }}">this link</a>.</p>

{%- if email %}
<p>You will be notified at {{ email }} when job results are available.</p>
{%- endif %}
{% endblock %}
```

Results page

The results page is used to display the results of a job, and is implemented by providing a `results` function. The function can either display the job results directly, or it can display links to allow output files to be downloaded. In the latter case, URLs to these files can be generated by calling *CompletedJob.get_results_file_url()*.

The example below assumes the backend generates a single output file on success, `output.pdb`, and a log file, `log`, on failure. It uses the utility function *saliweb.frontend.get_completed_job()* to get a *CompletedJob* object from the URL (this will show an error message if the job has not completed, or the password is invalid) and then looks for files in the job directory using the *CompletedJob.get_path()* method:

```
@app.route('/job/<name>')
def results(name):
    job = saliweb.frontend.get_completed_job(name,
                                             flask.request.args.get('passwd'))
```

(continues on next page)

(continued from previous page)

```
    # Determine whether the job completed successfully
    if os.path.exists(job.get_path('output.pdb')):
        template = 'results_ok.html'
    else:
        template = 'results_failed.html'
    return saliweb.frontend.render_results_template(template, job=job)
```

This also uses the function *saliweb.frontend.render_results_template()*, which as before will either display an HTML page (similarly to Flask's render_template), or XML in the case of automated usage.

On successful job completion, it shows the `results_ok.html` Jinja template, which uses the *CompletedJob.get_results_file_url()* method to show a link to download `output.pdb` and the *CompletedJob.get_results_available_time()* method to tell the user how long the results page will be available for:

```
{% extends "layout.html" %}

{% block body %}
<p>Job '<b>{{ job.name }}</b>' has completed.</p>

<p><a href="{{ job.get_results_file_url('output.pdb') }}">Download output PDB</a>.</p>

{{ job.get_results_available_time() }}
{% endblock %}
```

On failure it shows a similar page that links to the log file:

```
{% extends "layout.html" %}

{% block body %}
<p>Your job '<b>{{ job.name }}</b>' failed to produce any output models.</p>

<p>For more information, you can
<a href="{{ job.get_results_file_url('log') }}">download the log file</a>.
</p>
{% endblock %}
```

Results files

If individual results files can be downloaded, a `results_file` function should be provided. Similar to the results page, this looks up the job information using the URL, then sends it to the user using Flask's send_from_directory function. The user is prevented from downloading other files that may be present in the job directory, getting an HTTP 404 (file not found) error instead, using the Flask abort function:

```
@app.route('/job/<name>/<path:fp>')
def results_file(name, fp):
    job = saliweb.frontend.get_completed_job(name,
                                             flask.request.args.get('passwd'))
    if fp in ('output.pdb', 'log'):
        return flask.send_from_directory(job.directory, fp)
    else:
        flask.abort(404)
```

Note: The "results files" don't have to actually exist as real files in the job directory. Files can also be constructed on

the fly and their contents returned to the user in a custom Flask Response object.

Alternative submit/results page for short jobs

For short jobs, it may not be desirable for job submission to pop up a page containing a 'results' link that the user then needs to click on (as the job may be complete by that point). In this case, the job submission page can redirect straight to the job results page using *saliweb.frontend.redirect_to_results_page()*. To avoid the user seeing an uninformative 'job is still running' page, this page should be overridden using the `still_running_template` argument to *saliweb.frontend.get_completed_job()*. (Normally this would display something very similar to the submit page, but can auto-refresh if desired using *saliweb.frontend.StillRunningJob.get_refresh_time()*.) The resulting logic would look similar to:

```
@app.route('/job', methods=['GET', 'POST'])
def job():
    if flask.request.method == 'GET':
        return saliweb.frontend.render_queue_page()
    else:
        return submit_new_job()


@app.route('/job/<name>')
def results(name):
    job = get_completed_job(name, request.args.get('passwd'),
                            still_running_template='running.html')
    ...  # as for the previous results page, above


def submit_new_job():
    ...  # as for the previous submit page, above

    job.submit(email)

    # Go straight to the results page
    return saliweb.frontend.redirect_to_results_page(job)
```

It uses the following template as `running.html`, providing the `job` variable (which is a *saliweb.frontend.StillRunningJob* object), to notify the user and auto-refresh:

```
{% extends "layout.html" %}

{% block meta %}
<meta http-equiv="refresh" content="{{ job.get_refresh_time(10) }}" />
{% endblock %}

{% block body %}
<p>Your job {{ job.name }} has been submitted.</p>

<p>The job is currently running. When it is complete, results will be found
here - simply refresh or bookmark this page.</p>

{%- if job.email %}
<p>You will be notified at {{ job.email }} when the job has finished.</p>
{%- endif %}
{% endblock %}
```

See LigScore at https://github.com/salilab/ligscore/ for a web service that uses this submit/results logic.

Additional pages

Additional pages can be added if desired, simply by adding more Python functions with appropriate URLs (and adding the names of the functions to the `get_navigation_links` function in the `layout.html` Jinja template). Typically these just use `render_template` to show some content:

```
@app.route('/contact')
def contact():
    return flask.render_template('contact.html')


@app.route('/help')
def help():
    return flask.render_template('help.html')
```

1.4.4 Controlling page access

By default, all pages can be viewed by both anonymous and logged-in users. This can be modified, for example to restrict access only to named users, by checking the value of `flask.g.user` in any function, which is either a *LoggedInUser* object or `None`.

If access should be denied, either raise an *AccessDeniedError* exception (which will return an error page to the user with a message) or call flask.abort with a suitable HTTP error code, e.g. 401, "unauthorized" (which will return a generic error page without a message).

1.5 Deploying the web service

To actually deploy the web service, it is necessary to package the Python classes that implement the backend and frontend, then use the build system to install these classes in the correct location, together with other resources such as images, style sheets or text files needed by the web interface.

1.5.1 Prerequisites

Every service needs some basic setup:

- The service needs its own MySQL database, and two MySQL users set up, one for the backend and the other for the frontend. A sysadmin can set this up on the *modbase* machine.
- The service needs its own user on the *modbase* machine; for example, there is a *modloop* user for the ModLoop service. It is this user that runs *scons* (below). All of the backend also runs as this user, and jobs on the SGE clusters also run under this user's account. (It is not a good idea to use a regular user for this purpose, as it will use up the regular user's disk and runtime quota on the cluster, and bugs in the service could lead to deletion of that user's files or their exposure to outside attack.) A sysadmin can also set up this user account.
- The web service user needs a directory on the NetApp disk in order to store running jobs, and at least one directory on a local *modbase* disk so the frontend can create incoming jobs.
- A sysadmin needs to configure the web server on *modbase* so that the web service files are visible to the outside world. They can also password protect the page if it is not yet ready for a full release.
- It is usually a good idea to put the implementation files for a web service on GitHub, or in an SVN repository.

1.5.2 Quick start

The easiest way to set up a new web service is to have a sysadmin run the `make_web_service` script on the *modbase* machine. Given the name of the web service it will set up all the necessary files used for a basic web service. Run `make_web_service` with no arguments for further help.

Note: `make_web_service` should be run on a local disk (**not** /wynton). Most users on *modbase* have their home directories on a local disk, so this is generally OK by default. Note that the home directory should be accessible by the backend user in order for the build system to work; running `chmod a+rx ~` should usually be sufficient.

Example usage

For example, the user 'bob' wants to set up a web service for peptide docking.

1. He first chooses a "human readable" name for his service, "Peptide Docking". This name will appear on web pages and in emails, but can be changed later by editing the configuration file, if desired.
2. He also chooses a "short name" for his service, "pepdock". The short name should be a single lowercase word; it is used to name system and MySQL users, the Perl and Python modules, etc. It is difficult to change later, but is never seen by end users so is essentially arbitrary.
3. He asks a sysadmin to set up the web service, giving him or her the "short name" and the human readable name. (The sysadmin will run the *make_web_service* script.)
4. Bob can then get the web service from git or Subversion by running:

```
$ git clone git@github.com:salilab/pepdock.git [git]
$ svn co https://svn.salilab.org/pepdock/trunk pepdock [Subversion]
$ cd pepdock/conf
$ sudo -u pepdock cat ~pepdock/service/conf/backend.conf > backend.conf
$ sudo -u pepdock cat ~pepdock/service/conf/frontend.conf > frontend.conf
```

5. Bob edits the *configuration file* in `conf/live.conf` to adjust install locations, etc. if necessary, and fills in the template Python modules for the *backend* and *frontend*, in `backend/pepdock/__init__.py` and `frontend/pepdock/__init__.py`, respectively.
6. He writes test cases for both the frontend and backend (see *Testing*) and runs them to make sure they work by typing *scons test* in the pepdock directory.
7. He deploys the web service by simply typing *scons* in the pepdock directory. This will give him further instructions to complete the setup (for example, providing a set of MySQL commands to give to a sysadmin to set up the database).
8. Once deployment is successful, he asks a sysadmin to set up the web server on *modbase* so that the URL given in *urltop* in `conf/live.conf` works, and to register the service with Google Analytics. The resulting UA number should also get entered into the configuration file.
9. Whenever Bob makes changes to the service in his *pepdock* directory, he simply runs *scons test* to make sure the changes didn't break anything, then *scons* to update the live copy of the service, then *git commit* and *git push* to publish the changes at GitHub. (The backend will also need to restarted when he does this, but *scons* will show a suitable command line to achieve this.)
10. If Bob wants to share development of the service with another user, Joe, they should ask a sysadmin to give Joe *sudo* access to the *pepdock* account. Joe can then set up his own *pepdock* directory by cloning the repository from GitHub and then developing in the same way as Bob, above.

Note: Development of the service should generally be done by the regular ('bob') user; only the backend itself runs as the backend ('pepdock') user. Bob can however run any command as the 'pepdock' user using 'sudo' (e.g. `sudo -u pepdock scons` to run scons as the pepdock user). Note that sudo will ask for the regular user's (Bob's) password, not the pepdock account (which does not have a password anyway, and cannot be logged into). For advanced access, a shell can be opened as the backend user by running something like `sudo -u pepdock bash`.

1.5.3 Design tips

When designing a web service, the following design tips may be useful:

- The web service should implement little or none of the actual algorithm; instead, the algorithm should be implemented in another package that can be used independently. This allows others to use your algorithm on their own machines, rather than having to use Sali lab resources via the web service. The web service itself should only handle generating input files and nicely presenting any results (e.g. with interactive plots or protein structures). For example, ModLoop relies on MODELLER for the actual algorithm, while the algorithm used by the AllosMod web service is implemented in a separate AllosMod library, which allows the AllosMod protocol to be run from a command line.
- A web service must be self contained. If you absolutely must use external scripts in your web service, don't put them in your home directory or some other random place on the disk. Include and install them with the rest of the web service. See the MultiFoXS service for an example (in that case the external scripts are put in a *scripts* directory and installed in a cluster-accessible location).
- Web service dependencies must be well defined. If you need to use external software, like IMP, scikit, or gnuplot, don't compile your own version of that software and install it in a random place. Use "*module load*" to load the module for that software instead (if a module isn't available, ask a sysadmin to build one for you).

The following sections describe the various components of a web service in more detail, for developers that wish to set things up themselves without using the convenience scripts.

1.5.4 Backend Python package

The backend for the service should be implemented as a Python package in the `backend` subdirectory. Its name should be the same as the service, except that it should be all lowercase, and any spaces in the service name should be replaced with underscores. For example, the 'ModFoo' web service should be implemented by the file `backend/modfoo/__init__.py`). This package should implement a *Job* subclass and may also optionally implement *Database* or *Config* subclasses. It should also provide a function *get_web_service* which, given the name of a configuration file, will instantiate a *WebService* object, using these custom subclasses, and return it. This function will be used by utility scripts set up by the build system to run and maintain the web service. An example, building on previous ones, is shown below.

```
import saliweb.backend
import glob

class Database(saliweb.backend.Database):
    def __init__(self, jobcls):
        saliweb.backend.Database.__init__(self, jobcls)
        self.add_field(saliweb.backend.MySQLField('number_of_pdbs', 'INTEGER'))


class Job(saliweb.backend.Job):
    runnercls = saliweb.backend.WyntonSGERunner

```

(continues on next page)

(continued from previous page)

```
    def preprocess(self):
        pdbs = glob.glob("*.pdb")
        self._metadata['number_of_pdbs'] = len(pdbs)

    def run(self):
        script = """
for f in *.pdb; do
  grep '^HETATM' $f > $f.het
done
"""
        r = self.runnercls(script)
        r.set_options('-l diva1=1G')
        return r


def get_web_service(config_file):
    db = Database(Job)
    config = saliweb.backend.Config(config_file)
    return saliweb.backend.WebService(config, db)
```

1.5.5 Frontend Python package

The frontend for the service should be implemented as a Python package in the `frontend` subdirectory, named as for the *backend* (e.g. the 'ModFoo' web service's frontend should be implemented by the file `frontend/modfoo/__init__.py`). An example is shown below. For clarity, only the methods are shown, not their contents; for full implementations of the methods see the *Frontend* page.

```
from flask import render_template, request
import saliweb.frontend


app = saliweb.frontend.make_application(__name__)


@app.route('/')
def index():
    return render_template('index.html')


@app.route('/job', methods=['GET', 'POST'])
def job():
    # submit new job or show all jobs (queue)


@app.route('/job/<name>')
def results(name):
    # show results page


@app.route('/job/<name>/<path:fp>')
def results_file(name, fp):
    # download results file
```

1.5.6 Configuration file

The service's configuration should be placed in a configuration file in the `conf` subdirectory. Multiple files can be created if desired, for example to maintain both a testing and a live version of the service. Each configuration file can specify a different install location, MySQL database, etc. This directory will also contain the supplementary configuration files that contain the usernames and passwords that the backend and frontend need to access the MySQL database. Since these files contain sensitive information (passwords), they should **not** be group- or world-readable (*chmod 0600 backend.conf*), and if using SVN or git, **do not** put these database configuration files into the repository.

1.5.7 Using the build system

The build system is a set of extensions to SCons that simplifies the setup and installation of a web service. To use, create a directory in which to develop the web service, and create a file *SConstruct* in that directory similar to the following:

```
import saliweb.build

v = Variables('config.py')
env = saliweb.build.Environment(v, ['conf/live.conf', 'conf/test.conf'])
Help(v.GenerateHelpText(env))

env.InstallAdminTools()

Export('env')
SConscript('backend/modfoo/SConscript')
SConscript('frontend/modfoo/SConscript')
```

This script creates an *Environment* object which will set up the web service using either the configuration file *live.conf* or the file *test.conf* in the *conf* subdirectory.

The *Environment* class derives from the standard SCons Environment class, but adds additional methods which simplify the setup of the web service. For example, the *InstallAdminTools()* method installs a set of command-line admin tools in the web service's directory (see below). *SConscript* files in subdirectories can use similar methods (such as *InstallPython()*) to set up the rest of the necessary files for the web service.

To test the web service, run *scons test* from the command line on the *modbase* machine (see *Testing*).

To actually install the web service, run *scons build=live* or *scons build=test* from the command line on the *modbase* machine, as the web service backend user, to install using either of the two configuration files listed in the example above. (If *scons* is run with no arguments, it will use the first one, *live.conf*.) Before actually installing any files, this will check to make sure things are set up for the web service to work properly - for example, that the necessary MySQL users and databases are present.

1.5.8 Command-line admin tools

The build system creates several command-line admin tools in the *bin* subdirectory under the web service's install directory. These can be run by the web service user to control the service itself and manipulate jobs in the system.

service.py

This tool is used to start, stop or restart the backend itself for the web service. This daemon performs all functions of the web service, waiting for jobs submitted by the web frontend and submitting them to the cluster, harvesting completed cluster jobs, and expiring old job results. The tool also has a *condstart* option which will only start the service if it is not already running (the regular *start* option will complain if the service is running).

resubmit.py

This tool will move one or more jobs from the **FAILED** state back to the **INCOMING** state. It is designed to be used to resubmit failed jobs once whatever problem with the web service that caused these jobs to fail the first time around has been resolved.

deljob.py

This tool will delete one or more jobs in a given state. It can be used to remove failed jobs from the system, or to purge information from the database on expired jobs. Jobs in other states (such as **RUNNING** or **COMPLETED**) can also be deleted, but only if the backend service is stopped first, since that service actively manages jobs in these states.

failjob.py

This tool will force one or more jobs into the **FAILED** state. This is useful if, for example, due to a bug in the backend, a job didn't work properly but went into the **COMPLETED** state. The backend service must first be stopped in order to use this tool.

delete_all_jobs.py

This tool will delete all of the jobs from the web service, so can be used to 'restore to factory settings'. It deletes the database table, and all the files in all the job directories (even extraneous files that do not correspond to jobs in the database). It should be used with caution, as this cannot be undone.

list_jobs.py

This tool will show all the jobs in the given state(s). It is helpful for internal web services that don't have an easily accessible queue web page.

1.5.9 Testing

Before the framework is put into production it should be tested to make sure it works correctly. There are two main types of tests that should be done:

- **Unit tests** test individual parts of the service to make sure they work in isolation.
- **System tests** test the service as a whole.

Unit tests

To test the frontend, make a *test/frontend* subdirectory and put one or more Python scripts there. Each script can use the functions and classes in the *saliweb.test* module, together with test functionality provided by the Flask framework, to create simple instances of the web frontend and test various methods given different inputs. For example, a script to test the index page might look like:

```
import unittest
import saliweb.test

# Import the modfoo frontend with mocks
modfoo = saliweb.test.import_mocked_frontend("modfoo", __file__,
```

(continues on next page)

(continued from previous page)

```
                                                '../../frontend')


class Tests(saliweb.test.TestCase):

    def test_index(self):
        """Test index page"""
        c = modfoo.app.test_client()
        rv = c.get('/')
        self.assertIn(b'ModFoo: Modeling using Foo', rv.data)


if __name__ == '__main__':
    unittest.main()
```

Then write an SConscript file in the same directory to actually run the scripts, using the `RunPythonFrontendTests()` method. This might look like:

```
Import('env')

env.RunPythonFrontendTests(Glob("*.py"))
```

To test the backend, make a *test/backend* subdirectory and put one or more Python scripts there. Each script should define a subclass of *saliweb.test.TestCase* and define one or methods starting with *test_* using standard Python *unittest* methods such as *assertEquals*. A number of other utility classes are also provided in the *saliweb.test* module.

For example, to test that the *archive()* method of the ModFoo service (*Simple job* example) really does gzip all of the PDB files, a test case like that below could be used:

```
from __future__ import print_function
import unittest
import modfoo
import saliweb.test
import os


class JobTests(saliweb.test.TestCase):
    """Check custom ModFoo Job class"""

    def test_archive(self):
        """Test the archive method"""
        # Make a ModFoo Job test job in ARCHIVED state
        j = self.make_test_job(modfoo.Job, 'ARCHIVED')
        # Run the rest of this testcase in the job's directory
        with saliweb.test.working_directory(j.directory):
            # Make a test PDB file and another incidental file
            with open('test.pdb', 'w') as f:
                print("test pdb", file=f)
            with open('test.txt', 'w') as f:
                print("text file", file=f)

            # Run the job's "archive" method
            j.archive()

            # Job's archive method should have gzipped every PDB file but not
```

(continues on next page)

(continued from previous page)

```
            # anything else
            self.assertTrue(os.path.exists('test.pdb.gz'))
            self.assertFalse(os.path.exists('test.pdb'))
            self.assertTrue(os.path.exists('test.txt'))


if __name__ == '__main__':
    unittest.main()
```

Then write an SConscript file in the same directory to actually run the scripts, using the *RunPythonTests()* method. This might look like:

```
Import('env')

env.RunPythonTests(Glob("*.py"))
```

Run *scons test* to actually run the tests.

System tests

There is currently no rigorous way to carry out system tests other than *deploying the service*, then using the web interface to submit a job.

1.5.10 Examples

A simple example of a complete web service is ModLoop. The source code for this service can be found at https://github.com/salilab/modloop/ and the service can be seen in action at https://salilab.org/modloop/

1.6 Using the web service

This section covers the use of the web service by an end user. Here the 'ModFoo' example web service is used, which is defined by the *configuration file* to be deployed at *https://modbase.compbio.ucsf.edu/modfoo*.

1.6.1 Web interface

End users can simply use the web service by pointing their web browsers to *https://modbase.compbio.ucsf.edu/modfoo/*. This will display the index page, from where they can submit jobs or navigate to other pages.

1.6.2 User authentication

The web framework ties in automatically to Ursula's login page for web service accounts. If a user logs in, his or her email address is automatically supplied to forms that ask for it; job results pages are linked from the queue page; and, potentially, data files can be passed between multiple lab web services. Unauthenticated (anonymous) users can still use lab web services, however, unless the service is experimental or very expensive in terms of computer time.

1.6.3 Automated use

The web framework automatically sets up each web service to allow automation. This allows a program to submit jobs, check for completion, and finally to obtain the generated results. This is achieved with a REST-style interface that talks a simple form of XML. This interface is provided by the '/job' endpoint; for the fictional ModFoo web service this would be found at *https://modbase.compbio.ucsf.edu/modfoo/job*. XML output is requested by setting the HTTP Accept header to `application/xml`. Jobs can be submitted by sending an HTTP POST request to this URL containing the same form data that would be sent to the regular submit page. On successful submission, the returned XML file will contain a URL pointing to the job results. This URL can be queried by an HTTP GET to see if the job has finished (again with an Accept header). If it has, a list of URLs for job results files is returned; if it has not, an HTTP error is returned and the request can be retried later. Finally, the job results files can be downloaded using the provided URLs.

There is a simple Python interface to all Sali Lab services that use the framework, available on all Sali lab machines, that takes care of XML parsing and error handling for you. It can be used to submit jobs and collect results either from the command line or from other Python scripts. For example, to run a job on the fictional ModFoo service and wait for results, you can run from the command line on a lab machine something like:

```
module load web_service

web_service.py run https://modbase.compbio.ucsf.edu/modfoo/job \
                   input_pdb=@input.pdb job_name=testjob
```

Use `web_service.py help` for full details on using this utility.

Alternatively, you can do the same thing from Python with a script like:

```
#!/usr/bin/python

from saliweb import web_service

results = web_service.run_job('https://modbase.compbio.ucsf.edu/modfoo/job',
                              ['input_pdb=@input.pdb', 'job_name=testjob'])
```

(Note that you can also submit jobs to a web service from another one using the *SaliWebServiceRunner* class.)

Although */usr/bin/web_service.py* is only installed on Sali lab machines, you can download a copy and run it on any machine that has network access and has Python and curl installed.

Reference:

1.7 Module reference

For reference, the Python and Perl modules that provide the functionality are documented here.

1.7.1 The `saliweb.backend` Python module

class saliweb.backend.**Config**(*fh*)

This class holds configuration information such as directory locations, etc. *fh* is either a filename or a file handle from which the configuration is read.

admin_email

The email address of the admin.

populate(*config*)

Populate data structures using the passed *config*, which is a `configparser.ConfigParser` object.

This can be overridden in subclasses to read additional service-specific information from the configuration file.

send_admin_email(*subject*, *body*)
Send an email to the admin for this web service, with the given *subject* and *body*.

send_email(*to*, *subject*, *body*)
Send an email to the given user or list of users (*to*), with the given *subject* and *body*. *body* can either be the text itself, or an object from the Python email module (e.g. MIMEText, MIMEMultipart).

class saliweb.backend.**Database**(*jobcls*)
Management of the job database. Can be subclassed to add extra columns to the tables for service-specific metadata, or to use a different database engine. *jobcls* should be a subclass of *Job*, which will be used to instantiate new job objects.

add_field(*field*)
Add a new field (typically a *MySQLField* object) to each table in the database. Usually called in the constructor or immediately after creating the *Database* object.

set_track_hostname()
Add extra fields to support tracking the user's hostname

class saliweb.backend.**MySQLField**(*name*, *type*, *null=True*, *key=None*, *default=None*, *index=False*)
Description of a single field in a MySQL database. Each field must have a unique *name* (e.g. 'user') and a given *type* (e.g. 'VARCHAR(15)'). *null* specifies whether the field can be NULL (valid values are True, False, 'YES', 'NO'). *key* if given specifies what kind of key it is (e.g. 'PRIMARY' or 'PRI'). *default* specifies the default value of the field. If *index* is True, create an index on this field (the index gets the same name as the field, except with an '_index' suffix).

get_schema()
Get the SQL schema needed to create a table containing this field.

class saliweb.backend.**Runner**
Base class for runners, which handle the actual running of a job, usually on an SGE cluster (see the *WyntonSGERunner* subclass). To create a subclass, you must implement both a _run method and a _check_completed class method, set the _runner_name attribute to a unique name for this class, and call *Job.register_runner_class()* passing this class.

class saliweb.backend.**ClusterRunner**(*script*, *interpreter='/bin/sh'*)
Base class to run a set of commands on a compute cluster. Use a subclass specific to the cluster you want to use, such as *WyntonSGERunner*.

To use, pass a string *script* containing a set of commands to run, and use *interpreter* to specify the shell (e.g. */bin/sh*, */bin/csh*) or other interpreter (e.g. */usr/bin/python*) that will run them. These commands will be automatically modified to update a job state file at job start and end, if your interpreter is */bin/sh*, */bin/csh*, */bin/bash* or */bin/tcsh*. If you want to use a different interpreter you will need to manually add code to your script to update a file called job-state in the working directory, to contain just the simple text "STARTED" (without the quotes) when the job starts and just "DONE" when it completes.

Once done, you can optionally call *set_options()* to set command line options (e.g. time limits, requested resources) and/or *set_name()* to set the job name.

set_name(*name*)
Set the job name (equivalent to SGE's qsub -N option, or SLURM's sbatch -J option). If the name is not a valid name (e.g. SGE job names cannot start with a digit) then it is mapped to one that is.

set_options(*opts*)
Set the options to use, as a string, for example '-l mydisk=1G -p 0'. These will be specific to the queuing system (e.g. SGE, SLURM) you are using. Note that if you want to set the job name (SGE's -N, SLURM's -J option) it is better to use *set_name()*.

class saliweb.backend.**SGERunner**(*script*, *interpreter='/bin/sh'*)
Base class to run a set of commands on an SGE cluster. Use a subclass specific to the cluster you want to use, such as *WyntonSGERunner*.

See *ClusterRunner* for more information.

set_name(*name*)
Set the job name (equivalent to SGE's qsub -N option, or SLURM's sbatch -J option). If the name is not a valid name (e.g. SGE job names cannot start with a digit) then it is mapped to one that is.

set_sge_name(*name*)
Set the job name (equivalent to SGE's qsub -N option, or SLURM's sbatch -J option). If the name is not a valid name (e.g. SGE job names cannot start with a digit) then it is mapped to one that is.

set_sge_options(*opts*)
Set the options to use, as a string, for example '-l mydisk=1G -p 0'. These will be specific to the queuing system (e.g. SGE, SLURM) you are using. Note that if you want to set the job name (SGE's -N, SLURM's -J option) it is better to use *set_name()*.

class saliweb.backend.**WyntonSGERunner**(*script*, *interpreter='/bin/sh'*)
Run commands on the Wynton SGE cluster.

See *SGERunner* for more details.

class saliweb.backend.**SLURMRunner**(*script*, *interpreter='/bin/sh'*)
Base class to run a set of commands on a SLURM cluster (experimental).

See *ClusterRunner* for more information.

set_name(*name*)
Set the job name (equivalent to SGE's qsub -N option, or SLURM's sbatch -J option). If the name is not a valid name (e.g. SGE job names cannot start with a digit) then it is mapped to one that is.

class saliweb.backend.**LocalRunner**(*cmd*)
Run a program (given as a list of arguments or a single string) on the local machine.

The program must create a file called job-state in the working directory, to contain just the simple text "STARTED" (without the quotes) when the job starts and just "DONE" when it completes.

class saliweb.backend.**SaliWebServiceRunner**(*url*, *args*)
Run a job on another Sali lab web service. When the job completes, the resulting files are passed to the *Job.postprocess()* method, as a list of *SaliWebServiceResult* objects, where they can be downloaded if desired. This uses the web service's REST interface (see *Automated use* for further details). For example, to submit to the fictional ModFoo service, use something like:

```
r = SaliWebServiceRunner('http://modbase.compbio.ucsf.edu/modfoo/job',
                         ['input_pdb=@input.pdb', 'job_name=testjob'])
```

class saliweb.backend.**SaliWebServiceResult**(*url*)
Represent a single file, resulting from a *SaliWebServiceRunner* job.

download(*fh=None*)
Download the result file. If *fh* is given, it should be a Python file-like object, to which the file is written. Otherwise, the file is written into the job directory with the same name.

get_filename()
Return the name of the file corresponding to this result.

class saliweb.backend.**DoNothingRunner**
Do nothing, i.e. have the job complete immediately.

class saliweb.backend.**WebService**(*config*, *db*)

Top-level class used by all web services. Pass in a *Config* (or subclass) object for the *config* argument, and a *Database* (or subclass) object for the *db* argument.

create_database_tables()

Create all tables in the database used to hold job state.

do_all_processing(*daemonize=False*, *status_fh=None*)

Process incoming jobs, completed jobs, and old jobs. This method will run forever, looping over the available jobs, until the web service is killed. If *daemonize* is True, this loop will be run as a daemon (subprocess), so that the main program can continue. If *status_fh* is specified, it is a file handle to which "OK" is printed on successful startup.

drop_database_tables()

Drop all tables in the database used to hold job state.

get_job_by_name(*state*, *name*)

Get the job with the given name in the given job state. Returns a *Job* object, or None if the job is not found.

get_running_pid()

Return the process ID of a currently running web service, by querying the state file. If no service is running, return None; if the last run of the service failed with an unrecoverable error, raise a *StateFileError*.

version = None

Version number of the service, or None.

class saliweb.backend.**Job**(*db*, *metadata*, *state*)

Class that encapsulates a single job in the system. Jobs are not created by the user directly, but by querying a *WebService* object.

logger

A standard Python logger object (i.e it provides methods such as *debug*, *info*, *warning*) that can be used by job methods such as *run()* (but not *expire()*). By default this logger logs to a file called 'framework.log' in the job directory; the file is created when the first log message is emitted. See also *get_log_handler()*.

admin_fail(*email*)

Force a job into the FAILED state. This is intended to be used by the server administrator (via the failjob.py utility) to fail jobs which have erroneously completed.

archive()

Do any necessary processing when an old completed job reaches its archive time. Does nothing by default, but can be overridden by the user to compress files, etc. This method should not be called directly.

complete()

This method is called after a job completes. Does nothing by default, but can be overridden by the user. Note that it is rare to override this method - usually *finalize()* makes more sense. This method should not be called directly.

config

Config object (read-only)

delete()

Delete the job directory and database row.

directory

Current job working directory (read-only)

expire()

Do any necessary processing when an old completed job reaches its expire time; it is called just before the

job directory is deleted. Does nothing by default, but can be overridden by the user to mail the admin, etc. This method should not be called directly.

finalize()
Do any necessary finalizing when the job completes successfully. Does nothing by default. This is like *postprocess()* except that it cannot schedule any more jobs (*reschedule_run()*), i.e. it is only called on the last cycle of a multi-job run. This method should not be called directly.

get_log_handler()
Create and return a standard Python log Handler object. By default it directs log messages to a file called 'framework.log' in the job directory. This can be overridden to send log output elsewhere, e.g. in an email. Do not call this method directly; instead use *logger* to access the logger object.

name
Unique job name (read-only)

postprocess(*results=None*)
Do any necessary postprocessing when the job completes successfully. Does nothing by default. Note that a user-defined postprocess method can call *reschedule_run()* to request that the backend runs a new cluster job if necessary. *results*, if given, contains explicit results from the job runner. For example, the *SaliWebServiceRunner* returns a list of files generated by the web service (as *SaliWebServiceResult* objects). See also *finalize()*. This method should not be called directly.

preprocess()
Do any necessary preprocessing before the job is actually run. Does nothing by default. Note that a user-defined preprocess method can call *skip_run()* to skip running of the job on the cluster, if it is determined in preprocessing that a job run is not necessary. This method should not be called directly.

classmethod register_runner_class(*runnercls*)
Maintain a mapping from names to *Runner* classes. If you define a *Runner* subclass, you must call this method, passing that subclass.

rerun(*data*)
Run a rescheduled job (if *postprocess()* called *reschedule_run()* to run a new job). *data* is a Python object passed from the *reschedule_run()* method. Like *run()*, this should create and return a suitable *Runner* instance.

By default, this method simply discards *data* and calls the regular *run()* method. You can redefine this method if you want to do something different for rescheduled runs.

reschedule_run(*data=None*)
Tell the backend to schedule another job to be run on the cluster once postprocessing is complete (the job moves from the POSTPROCESSING state back to RUNNING).

It is only valid to call this method from the POSTPROCESSING state, usually from a user-defined *postprocess()* method.

The rescheduled job is run by calling the *rerun()* method, which is passed the *data* Python object (if any).

Note that because the rescheduled job itself will be postprocessed once finished, you must be careful not to create an infinite loop here. This could be done by using a file in the job directory or a custom field in the job database to prevent a job from being rescheduled more than a certain number of times, and/or to pass state to the *postprocess()* method (so that it knows it is postprocessing a rescheduled job rather than the first job).

resubmit()
Make a FAILED job eligible for running again.

run()
Run the job, e.g. on an SGE cluster. Must be implemented by the user for each web service; it should create and return a suitable *Runner* instance. For example, this could generate a simple script and pass it to an *SGERunner* instance. This method should never be called directly; it is automatically called by the backend when needed. To run a new job, call *reschedule_run()* instead.

send_job_completed_email()
Email the user (if requested) to let them know job results are available. (It does this by calling *send_user_email()*.) Can be overridden to disable this behavior or to change the content of the email.

send_user_email(*subject*, *body*)
Email the owner of the job with the given *subject* and *body*. (If no email address was given by the user, this does nothing.)

service_name
Web service name (read-only)

skip_run()
Tell the backend to skip the actual running of the job, so that when preprocessing has completed, it moves directly to the COMPLETED state, skipping RUNNING, POSTPROCESSING, and FINALIZING.

It is only valid to call this method from the PREPROCESSING state, usually from a user-defined *preprocess()* method.

url
URL containing job results (read-only)

Exceptions

exception saliweb.backend.**ConfigError**
Exception raised if a configuration file is inconsistent.

exception saliweb.backend.**InvalidStateError**
Exception raised for invalid job states.

exception saliweb.backend.**RunnerError**
Exception raised if the runner (such as SGE) failed to run a job.

exception saliweb.backend.**SanityError**
Exception raised if a new job fails the sanity check, e.g. if the frontend added invalid or inconsistent information to the database.

exception saliweb.backend.**StateFileError**
Exception raised if a previous run is still running or crashed.

1.7.2 The saliweb.build Python module

This module provides a simple SCons-based build infrastructure for web services.

class saliweb.build.**Environment**(*variables*, *configfiles*[, *version*[, *service_module*]])
A simple class based on the standard SCons Environment class. This class should be instantiated in the top-level SConstruct file of a web service, and will check the configuration and install basic files. It must be given a set of SCons Variables, and the name of one or more configuration files. The user will be able to choose which configuration to build with by specifying the *build* option on the command line; if the user does not give this option, the first configuration file in *configfiles* is used. If *version* is specified, it should be a string specifying the version number of the service (e.g. "1.0.0"). If it is not, it is assumed that the web service is being deployed from an SVN checkout, and the `svnversion` program is run to attempt to determine the version number.

If *service_module* is specified, it is the name of the Python and Perl modules used to implement the service (it must be lowercase and contain no spaces). If not given, it is generated automatically from the service name in the configuration file (the name is lowercased and spaces are replaced with underscores).

InstallAdminTools ([*tools*])
Installs command-line admin tools in the `bin` directory underneath the installation directory. *tools* is a list of names to install that must be selected from the convenience modules provided by the *saliweb.backend* package, such as *resubmit* or *service*. If *tools* is not specified, all tools are installed.

InstallCGIScripts ([*scripts*])
Installs CGI scripts that control the frontend, in the `cgi` directory underneath the installation directory. *scripts* is a list of names to install that must be selected from the various display_*_page() methods implemented by the *saliweb::frontend* class, such as *index.cgi* or *submit.cgi*. If *scripts* is not specified, all scripts are installed.

InstallPython (*files*[, *subdir*])
Installs a provided list of Python files in the `python` directory underneath the installation directory. If installing subpackages, also specify the *subdir* argument to install them in a subdirectory.

InstallHTML (*files*[, *subdir*])
Installs a provided list of static files (usually HTML, although any static resource, such as images, can be installed) in the `html` directory underneath the installation directory. The files can be installed in a subdirectory if desired by giving the *subdir* argument.

InstallPythonFrontend (*files*[, *subdir*])
Installs a provided list of Python frontend files in the `frontend` directory underneath the installation directory. If installing subpackages, also specify the *subdir* argument to install them in a subdirectory.

InstallFrontend (*files*[, *subdir*])
Installs a provided list of frontend support files (generally Jinja2 templates) in the *frontend'* directory underneath the installation directory. The files can be installed in a subdirectory (such as `templates`) if desired by giving the *subdir* argument.

InstallCGI (*files*[, *subdir*])
Installs a provided list of CGI scripts in the `cgi` directory underneath the installation directory. The files can be installed in a subdirectory if desired by giving the *subdir* argument. This is only required if you need to install additional CGI scripts; in most cases, the *InstallCGIScripts()* method installs all the needed scripts.

InstallPerl (*files*[, *subdir*])
Installs a provided list of Perl modules in the `lib` directory underneath the installation directory. The files can be installed in a subdirectory if desired by giving the *subdir* argument.

InstallTXT (*files*[, *subdir*])
Installs a provided list of text files in the `txt` directory underneath the installation directory. The files can be installed in a subdirectory if desired by giving the *subdir* argument.

RunPerlTests (*tests*)
Runs a set of Perl tests of the frontend implementation.

RunPythonTests (*tests*)
Runs a set of Python tests of the backend implementation.

class Frontend (*name*)
This class is used to install an alternative frontend called *name*. There must be a corresponding section in the configuration file, and a Perl module, for this frontend (for example, a frontend called `foo` needs a section in the configuration file called `[frontend:foo]` and a Perl module (installed with *Environment.InstallPerl()*) called `foo.pm`). Methods are provided to install files for the frontend. They function identically to the methods in the *Environment* class, but install the files in a subdirectory of the web service called *name*.

InstallHTML (*files*[, *subdir*])

InstallTXT (*files*[, *subdir*])

InstallCGIScripts ([*scripts*])

1.7.3 The saliweb.frontend Python module

class saliweb.frontend.**CompletedJob**(*sql_dict*)

A job that has completed. Use *get_completed_job()* to create such a job from a URL.

get_path(*fname*)

Get the full path to a file in the job's directory.

Parameters **fname** (*str*) – The file name

Returns Full path to the file in the job's directory.

get_results_available_time()

Get an HTML fragment stating how long results will be available

get_results_file_url(*fname*)

Return a URL which the user can use to download the passed file. The file must be in the job directory (or a subdirectory of it); absolute paths are not allowed. If files are compressed with gzip, the .gz extension can be ommitted here if desired. (If it is ommitted, the file will be automatically decompressed when the user downloads it; otherwise the original .gz file is downloaded.)

class saliweb.frontend.**LoggedInUser**(*name*, *rd*)

Information about the logged-in user. *g.user* is set to an instance of this class, or None if no user is logged in.

email = None

The contact email address of the user

first_name = None

The first name of the user

institution = None

The user's institution

last_name = None

The last name of the user

modeller_key = None

The user's MODELLER license key

name = None

The login name of the user

class saliweb.frontend.**IncomingJob**(*given_name=None*)

Represents a new job that is being submitted to the backend. Each new job has a unique name and a directory into which input files can be placed. Once all input files are in place, *submit()* should be called to submit the job to the backend.

Parameters **given_name** (*str*) – A user-provided name for the job.

directory = None

The directory on disk for this job. Input files should be placed in this directory prior to calling *submit()*.

get_path(*fname*)

Get the full path to a file in the job's directory.

Parameters **fname** (*str*) – The file name

Returns Full path to the file in the job's directory.

name = None
The name of the job. Note that this is not necessarily the same as the name given by the user, since it must be unique, and fit in our database schema. (The user-provided name is thus sanitized if necessary and a unique suffix added.)

results_url
The URL where this job's results will be found when it is complete. This is only filled in when *submit()* is called.

submit(*email=None*, *force_results_xml=False*)
Submits the job to the backend to run on the cluster. If an email address is provided, it is notified when the job completes. If *force_results_xml* is True, *force_xml=True* is passed to the results URL, which can be used to force XML output even without the HTTP Accept header being set (used for backwards compatibility).

class saliweb.frontend.**StillRunningJob**(*name*, *passwd*, *email*, *submit_time*)
A job that is still running. See the *still_running_template* argument to *get_completed_job()*.

email = None
Email address used to notify the user of job completion

get_refresh_time(*minseconds*)
Get a suitable time, in seconds, to wait to refresh the 'job is still running' page. It will be at least *minseconds*.

name = None
The name of the job

passwd = None
The password needed to access the job web pages

submit_time = None
The time (as a datetime.datetime object) when this job was submitted

class saliweb.frontend.**Parameter**(*name*, *description*, *optional=False*)
Represent a single parameter (with help). This is used to provide help to users of the REST API. See *make_application()*.

Parameters

- **name** (*str*) – The name (must match that of the form item).
- **description** (*str*) – Help text about the parameter and its use.
- **optional** (*bool*) – Whether the parameter can be omitted.

class saliweb.frontend.**FileParameter**(*name*, *description*, *optional=False*)
Represent a single file upload parameter (with help). See *Parameter*.

saliweb.frontend.**make_application**(*name*, *parameters=[]*, *static_folder='html'*, **args*, ***kwargs*)
Make and return a new Flask application.

Parameters

- **name** (*str*) – Name of the Python file that owns the app. This should normally be *__name__*.
- **parameters** (*list*) – The form parameters accepted by the 'submit' page. This should be a list of *Parameter* and/or *FileParameter* objects, and is used to provide help for users of the REST API.

Returns A new Flask application.

Note: Any additional arguments are passed to the Flask constructor.

saliweb.frontend.**get_completed_job**(*name*, *passwd*, *still_running_template=None*)
Create and return a new *CompletedJob* for a given URL. If the job is not valid (e.g. incorrect password) an exception is raised.

Parameters

- **name** (*str*) – The name of the job.
- **passwd** (*str*) – Password for the job.
- **still_running_template** (*str*) – If given, the name of a Jinja2 template that will be used to report the 'job is still running' error; it is passed the error message as `message` and a *StillRunningJob* object as `job`.

Returns A new CompletedJob.

Return type *CompletedJob*

saliweb.frontend.**get_db**()
Get the MySQL database connection

saliweb.frontend.**render_queue_page**()
Return an HTML list of all jobs. Typically used in the */job* route for a GET request.

saliweb.frontend.**check_email**(*email*, *required=False*)
Check a user-provided email address for sanity. If the address is invalid, raise an *InputValidationError* exception.

Parameters

- **email** (*str*) – The email address to check.
- **required** (*bool*) – If True, an empty email address will also result in an exception (usually it is recommended that the email address is optional).

saliweb.frontend.**check_modeller_key**(*modkey*)
Check a provided MODELLER key. If the key is empty or invalid, raise an *InputValidationError* exception.

Parameters **modkey** (*str*) – The MODELLER key to check.

saliweb.frontend.**check_pdb**(*filename*, *show_filename=None*)
Check that a PDB file really looks like a PDB file. If it does not, raise an *InputValidationError* exception.

Parameters

- **filename** (*str*) – The PDB file to check.
- **show_filename** (*str*) – If provided, include this filename in any error message to identify the PDB file (useful for services that allow upload of multiple PDB files).

saliweb.frontend.**pdb_code_exists**(*code*)
Return true iff the PDB code (e.g. 1abc) exists in our local copy of the PDB.

saliweb.frontend.**get_pdb_code**(*code*, *outdir*)
Look up the PDB code (e.g. 1abc) in our local copy of the PDB, and copy it into the given directory (usually an incoming job directory). The file will be named in standard PDB fashion, e.g. `pdb1abc.ent`. The full path to the file is returned. If the code is invalid or does not exist, raise an *InputValidationError* exception.

Parameters

- **code** (*str*) – The PDB code to access (e.g. 1abc)
- **outdir** (*str*) – The directory to copy the PDB file into

Returns The full path to the new file in `outdir`

saliweb.frontend.**get_pdb_chains**(*pdb_chain*, *outdir*)

Similar to *get_pdb_code()*, find a PDB in our database, and make a new PDB containing just the requested one-letter chains (if any) in the given directory. The PDB code and the chains are separated by a colon. (If there is no colon, no chains, or the chains are just '-', this does the same thing as *get_pdb_code()*.) For example, '1xyz:AC' would make a new PDB file containing just the A and C chains from the 1xyz PDB. The full path to the file is returned. If the code is invalid or does not exist, or at least one chain is specified that is not in the PDB file, raise an *InputValidationError* exception.

Parameters

- **pdb_chain** (*str*) – PDB code and chain IDs, separated by a colon
- **outdir** (*str*) – Directory to write the PDB file into

Returns Full path to the new PDB file

saliweb.frontend.**render_results_template**(*template_name*, *job*, *extra_xml_outputs=[]*, *extra_xml_metadata={}*, *extra_xml_links={}*, ***context*)

Render a template for the job results page. This normally functions like *flask.render_template* but will instead return XML if the user requests it (for the REST API). The XML file will include download links to any file mentioned in the template with *CompletedJob.get_results_file_url()*. Extra downloadable files can be added to the XML output by listing them in *extra_xml_outputs*. Custom tags can also be added to the XML output by listing them in *extra_xml_metadata*, which is a dict (keys are XML tag names, values are the XML values). *extra_xml_links* is similar except that the values are hyperlinks (xlink:href targets).

saliweb.frontend.**render_submit_template**(*template_name*, *job*, ***context*)

Render a template for the job submission page. This normally functions like *flask.render_template* but will instead return XML if the user requests it (for the REST API).

For very quick jobs that take only a few seconds to run, consider using *redirect_to_results_page()* instead.

saliweb.frontend.**redirect_to_results_page**(*job*)

Perform a redirect from the job-submission page to the job-results page. This normally functions like *flask.redirect*, but will instead return an XML document if the user requests it (for the REST API).

This can be used instead of *render_submit_template()* for a just-submitted job. (This is more appropriate for jobs that take only a few seconds to run.) The job results page should in turn call *get_completed_job()* with the `still_running_template` parameter to provide information on the job submission.

Parameters **job** (*IncomingJob*) – The just-submitted job.

exception saliweb.frontend.**InputValidationError**

Invalid user input, usually during a job submission. These errors are handled by reporting them to the user and asking them to fix their input accordingly.

exception saliweb.frontend.**AccessDeniedError**

Attempt by the user to access a protected page. These errors can be raised by any page and are generally handled by displaying an HTML/XML error page.

1.7.4 The saliweb.test Python module

Utility classes and functions to aid in testing Sali lab web services.

saliweb.test.**tmpdir**
The full path to the temporary directory.

class saliweb.test.**MockJob**(*name*, *directory*)
A temporary job for testing web service results pages. Create by calling *make_frontend_job()*.

directory = None
Temporary directory containing result files (see *make_file()*)

make_file(*name*, *contents=''*)
Make a file in the job's directory with the given contents

name = None
Job name

passwd = None
Password needed to construct URLs for this job

class saliweb.test.**TestCase**(*methodName='runTest'*)
Custom TestCase subclass for testing Sali web service backends

get_test_directory()
Get the full path to the directory containing test scripts. This can be useful for getting supplemental files needed by tests, which can be stored in a subdirectory of the test directory.

make_test_job(*jobcls*, *state*)
Make a test job of the given class in the given state (e.g. RUNNING, POSTPROCESSING) and return the new object. A temporary directory is created for the job to use (as Job.directory) and will be deleted automatically once the object is destroyed.

saliweb.test.**get_modeller_key**()
Get a valid mock Modeller key for testing

saliweb.test.**import_mocked_frontend**(*pkgname*, *test_file*, *topdir*)
Import the named frontend module (e.g. 'modloop'), and return it. This sets up the environment with mocked configuration so that the module can be tested without being installed and without a live database. For this to work properly, it should be called *before* importing *saliweb.frontend*.

Parameters

- **pkgname** (*str*) – The name of the web service frontend Python module to import.
- **test_file** (*str*) – File name of the test file (usually *__file__*).
- **topdir** (*str*) – Relative path from the test file to the top-level Python directory, i.e. that from which 'import pkgname' will work.

saliweb.test.**make_frontend_job**(*name*)
Context manager to make a temporary job. See *MockJob*. This can be used to test the job results page and the download of results files.

saliweb.test.**temporary_working_directory**()
Simple context manager to run in a temporary directory. While the context manager is active (within the 'with' block) the current working directory is set to a temporary directory. When the context manager exists, the working directory is reset and the temporary directory deleted.

saliweb.test.**working_directory**(*workdir*)
Context manager to temporarily set the working directory. While the context manager is active (within the 'with'

block) the current working directory is set to *workdir*. When the context manager exists, the working directory is reset.

1.7.5 The `saliweb::Test` Perl module

This module provides Perl functions and classes useful for testing the legacy Perl CGI web frontend.

class saliwebTest(*module_name*)
The main utility class used to test the frontend. *module_name* should be the same of the Perl module that implements the frontend.

make_frontend()
Make and return a new Perl frontend object (of the class given by *module_name* when this class was created).

1.7.6 The legacy `saliweb::frontend` Perl module

This module provides Perl functions and classes for implementing the web frontend using legacy CGI scripts. This *should not be used* for new web services, which should instead use Python Flask - see *saliweb.frontend*.

class saliwebfrontend(*config_file*, *version*, *server_name*)
The main class used by the frontend. Typically a web service will subclass this class and override one or more methods to actually implement the web interface. The class creates a Perl CGI object which can be used to actually output web pages, and provides methods that behave similarly to the corresponding methods in CGI.pm, such as *start_html()*.

htmlroot
The top-level URL under which all static web files (HTML, style sheets, images) are found (read-only).

cgiroot
The top-level URL under which all CGI scripts are found (read-only).

txtdir
The absolute path to the directory containing files installed with `InstallTXT()`.

http_status
The HTTP status code (e.g. 200 OK, 404 Not Found) that will be reported when the web page is printed.

version
The version number of this web service (read-only).

version_link
An HTML fragment that shows the version as a link to the github page for this web service, if *github* is set in the config file (if not, returns the same content as *version*). Read-only.

user_name
If a user is authenticated against the service, this is their user name; otherwise, it is undef (read-only).

email
If a user is authenticated against the service, this is their email address; otherwise, it is undef (read-only).

modeller_key
If a user is authenticated against the service and has already provided a MODELLER key, this is it; otherwise, it is undef (read-only).

cgi
A pointer to the CGI.pm object used to display HTML (read-only).

dbh
The database handle.

index_url
submit_url
queue_url
help_url
faq_url
links_url
about_url
news_url
contact_url
results_url
download_url
Absolute URLs to each web page (read-only).

get_header_page_title()
Return the HTML fragment used to display the page title inside a div in the page header. By default, this just displays the lab logo and the page title, but can be overridden if desired.

get_lab_navigation_links()
Return a reference to a list of lab resources and services, used by *get_header()*. This can be overridden in subclasses to add additional links.

get_navigation_links()
Return a reference to a list of navigation links, used by *get_header()*. This should be overridden for each service to add links to pages to submit jobs, show help, list jobs in the queue, etc.

get_project_menu()
Return an HTML fragment which will be displayed in a project menu, used by *get_header()*. This can contain general information about the service, links, etc., and should be overridden for each service. Usually, it is displayed on the left side of the web page. On very narrow screens (e.g. smart phones in portrait mode) it is omitted.

display_index_page()
display_submit_page()
display_queue_page()
display_help_page()
display_results_page()
display_download_page()
Convenience methods designed to be called from CGI scripts. Each displays a complete web page by calling *start_html()*, *get_header()*, *get_footer()*, and *end_html()*. The actual page content is obtained from a similarly named get_*_page() method; for example, *display_index_page()* calls *get_index_page()*. Each method also calls *check_page_access()* to check whether access to the page is permitted, and *get_page_is_responsive()* to determine whether the page is responsive (resizeable).

get_index_page()
Return the HTML content of the index page. This is empty by default, and must be overridden for each web service. Typically this will display a form for user input (multi-page input can be supported if intermediate values are passed between pages).

get_submit_page()
Return the HTML content of the submit page (that shown when a job is submitted to the backend). This is empty by default, and must be overridden for each web service. Typically this method will perform checks on the input data (throwing an *InputValidationError* to report any problems), then call *make_job()* and its own *submit()* method to actually submit the job to the cluster, then point the user to the URL where job results can be obtained.

get_results_page(*job*)
Return the HTML content of the results page (that shown when the user tries to view job results). It is passed a *CompletedJob* object that contains information such as the name of the job and the time at which job results will be removed, and is run in the job's directory. This method is empty by default, and must be overridden for each web service. Typically this method will display any job failures (e.g. log files), display the job results directly, or provide a set of links to allow result files to be downloaded (by calling *get_results_file_url()*).

get_queue_page()
Return the HTML content of the queue page. By default this simply shows all jobs in the queue in date order, plus some basic help text. (Note that there is currently no interface defined to do this any differently. If you need to customize the queue page, please talk to Ben so we can design a suitable interface.)

get_help_page(*type*)
Return the HTML content of help, contact, FAQ, links, about, or news pages; the passed *type* parameter will be *help*, *contact*, *faq*, *links*, *about*, or *news*. By default this simply displays a suitable text file installed as part of the web service in the `txt` directory, named `help.txt`, `contact.txt`, `faq.txt`, `links.txt`, `about.txt`, or `news.txt` respectively.

get_download_page()
Return the HTML content of the download page. This is empty by default.

check_page_access(*page_type*)
Check whether access to the given *page_type* is allowed. *page_type* is one of 'index', 'submit', 'queue', 'results', 'help', 'download'. It should simply return if access is allowed, or throw an *AccessDeniedError* exception if access is not permitted. By default, it simply returns, allowing all access, for all pages except the submit page, from which certain IPs are blocked.

get_page_is_responsive(*page_type*)
Returns true iff the given page is 'responsive', that is it can be safely resized to be much larger or smaller than the default size. Pages that are *not* responsive will be displayed at the default size, which doesn't look great on most mobile devices for example (the user will typically have to resize and/or pan the screen, or read very small text). A responsive page will be resized to fit the smartphone screen, which could also look bad if page elements aren't designed to scale.

page_type is as for `check_page_access()`. By default, the queue and help pages are marked as responsive. It can be overridden if other pages (such as the index page) are also resizable.

download_results_file(*job*, *file*)
This method is called to download a single results file (when the user follows a URL provided by *get_results_file_url()*), provided that *allow_file_download()* returns true. It is called in the job directory with a *CompletedJob* object and a relative path, and is expected to print out the HTTP header and then the contents of the file. By default, the method uses the MIME type returned by *get_file_mime_type()* in the header, then prints out the file if it physically exists on disk, or if it does not but a gzip-compressed version of it does (with .gz extension) it decompresses the file and prints that. This method can be overridden, for example to download other "files" which don't really exist on the disk.

allow_file_download(*file*)
When downloading a results file (see *download_results_file()*) this method is called to check whether the file is allowed to be downloaded, and should return true if it is. (For example, the job results directory may contain intermediate output files that should not be downloaded for efficiency or security reasons.) By default, this method always returns true.

get_file_mime_type(*file*)
When downloading a results file (see *download_results_file()*) this method is called to get the correct MIME type for the file. By default, it handles PNG and SVG images, and for all other files returns 'text/plain'. You may need to override this, for example, if some of your results files are tar files

('application/x-tar') or other types of image.

get_submit_parameter_help()
Return a reference to a list of parameters accepted by the submit page. This is used to document the REST web service, and should be overridden for each service. Each parameter should be the result of calling *parameter()* or *file_parameter()*.

parameter(*key*, *help*[, *optional*])
Represent a single parameter (with help), used as input to *get_submit_parameter_help()*. 'key' should match the name of the parameter used in the HTML form on the index page.

file_parameter(*key*, *help*[, *optional*])
Represent a single file upload parameter (with help), used as input to *get_submit_parameter_help()*.

make_job(*jobname*)
This creates and returns a new *IncomingJob* object that represents a new job, using a user-provided job name. The new job has its own directory into which input files can be placed, and once this is finished, *submit()* should be called to actually submit the job. This is typically used in *get_submit_page()*.

resume_job(*jobname*)
This creates and returns a *IncomingJob* object that represents an incoming job. This job must have been previously created using *make_job()*, and jobname must match the true name of that job (*saliweb::frontend.IncomingJob.name*) not the original user-provided name. This is used with multiple-page submissions, e.g. if the user must upload several files into the job directory before the job is submitted. Once done, *submit()* should be called to actually submit the job.

help_link(*target*)
Given an HTML anchor target, this returns an HTML fragment that creates a link to the help pages.

start_html([*style*])
Return the content of the head section of the web page, containing scripts, style sheets, and the title. If *style* is provided, this is the URL for a CSS style sheet; if not provided, a default Sali lab style is used. It usually does not make sense to override this method in derived classes (instead, override *get_start_html_parameters()*).

get_start_html_parameters(*style*)
Return a hash of arguments suitable for passing to CGI.pm's start_html() method. This can be overridden in derived classes, for example to add additional scripts or CSS style sheets.

end_html()
Return the content of the end of the web page.

get_header()
Return the header of each web page, which contains navigation links (provided by *get_navigation_links()*), a side menu for the service (provided by *get_project_menu()*), and links to other services.

get_footer()
Return the footer of each web page. By default, this is empty, but it can be subclassed to display references, contact addresses etc.

class saliweb::frontend.**IncomingJob**
This represents a new job that is being submitted to the backend. These objects are created by calling *make_job()*. Each new job has a unique name and a directory into which input files can be placed. Once all input files are in place, *submit()* should be called to submit the job to the backend.

name
The name of the job. Note that this is not necessarily the same as the name given by the user, since it must

be unique, and fit in our database schema. (The user-provided name is thus sanitized if necessary and a unique suffix added.)

directory
The directory on disk for this job. Input files should be placed in this directory prior to calling *submit()*.

results_url
The URL where this job's results will be found when it is complete. This is only filled in when *submit()* is called. Attempting to query this attribute before then will result in an *InternalError*.

submit ([*email*])
Submits the job to the backend to run on the cluster. If an email address is provided, it is notified when the job completes.

class saliweb::frontend.**CompletedJob**
This represents a job that has completed, and for which results are available. These objects are created automatically and passed to *saliwebfrontend.get_results_page()*, and can be queried to get information about the job.

name
The name of the job.

directory
The directory on disk containing job results.

results_url
The URL where this job's results can be found.

unix_archive_time
The Unix time (seconds since the epoch, in UTC) at which job results will become unavailable. (Use standard Perl functions such as `gmtime` and `strftime` to make this human-readable, or use *to_archive_time* or *get_results_available_time()* instead.) If the backend is configured to never archive job results, this will return undef.

to_archive_time
A human-readable string giving the time from now at which job results will become unavailable (e.g. '6 days', '24 hours'). If the backend is configured to never archive job results, or the time has already passed, this will return undef. See also *get_results_available_time()*.

get_results_available_time()
This will return a short paragraph, suitable for adding to a human-readable results page, indicating how long the results will be available for. If the backend is configured to never archive job results, or the time has already passed, this will simply return an empty string.

get_results_file_url (*file*)
Given a file which is an output file from the job, this will return a URL which can be used to download the file. The filename should be relative to the job directory, not an absolute path. The actual download of the file is handled by *download_results_file()*.

exception saliweb::frontend.**AccessDeniedError**(*message*)
This exception is raised if the end user does not have permission to view a page. It is generally raised from within *check_page_access()*.

exception saliweb::frontend.**InputValidationError**(*message*)
This exception is typically used to report failures with job submission (due to invalid user input) from within *get_submit_page()* or functions it calls. These errors are handled by reporting them to the user and asking them to fix their input accordingly.

exception saliweb::frontend.**InternalError**(*message*)
exception saliweb::frontend.**DatabaseError**(*message*)
These exceptions are used to report fatal errors in the frontend, such as an inability to create necessary directories

or files (e.g. the disk filled up), failure to connect to the MySQL database, etc. These errors are reported to the server admin so that they can fix the problem.

saliweb::frontend.**check_required_email**(*email*)
Check a provided email address. If the address is empty or is invalid, throw an *InputValidationError* exception.

saliweb::frontend.**check_optional_email**(*email*)
Check a provided email address. This is similar to *check_required_email()*, except that only invalid addresses cause an error; it is OK to provide an empty address.

saliweb::frontend.**check_modeller_key**(*modkey*)
Check a provided MODELLER key. If the key is empty or invalid, throw an *InputValidationError* exception.

saliweb::frontend.**pdb_code_exists**(*code*)
Return true iff the PDB code (e.g. 1abc) exists in our local copy of the PDB.

saliweb::frontend.**get_pdb_code**(*code*, *outdir*)
Look up the PDB code (e.g. 1abc) in our local copy of the PDB, and copy it into the given directory (usually an incoming job directory). The file will be named in standard PDB fashion, e.g. pdb1abc.ent. The full path to the file is returned. If the code is invalid or does not exist, throw an *InputValidationError* exception.

saliweb::frontend.**get_pdb_chains**(*code_and_chains*, *outdir*)
Similar to *get_pdb_code()*, find a PDB in our database, and make a new PDB containing just the requested one-letter chains (if any) in the given directory. The PDB code and the chains are separated by a colon. (If there is no colon, no chains, or the chains are just '-', this does the same thing as *get_pdb_code()*.) For example, '1xyz:AC' would make a new PDB file containing just the A and C chains from the 1xyz PDB. The full path to the file is returned. If the code is invalid or does not exist, or at least one chain is specified that is not in the PDB file, throw an *InputValidationError* exception.

saliweb::frontend.**sanitize_filename**(*filename*)
Make sure that a user-provided string is suitable for use as a filename component (this will remove things like spaces and relative paths). The sanitized version of the filename is returned.

1.8 Installation

1.8.1 In the Sali lab

There is no need for most users to install the web framework. It is already installed for you by our sysadmins on the 'modbase' server machine.

Sysadmins: if you do need to make modifications to the framework itself, this can be done by cloning the git repository to a directory on modbase, making your changes, writing test cases to exercise those changes, running 'scons test' to make sure the changes didn't break anything, 'git commit' and 'git push' to commit the changes, then running 'sudo scons install' to update the installed version of the framework.

1.8.2 Outside of the Sali lab

The framework is designed to work in the Sali lab environment, but can be used in other environments with some modification.

Prerequisites

- Python 3. The backend of the framework, which takes care of running jobs, is implemented in Python. (It does not work with Python 2.)
- Flask. The frontend, which handles user submission of jobs and the display of results, is implemented in Python 3, and uses the Flask microframework.
- SGE. The framework expects to be run on a machine that is a submit host to a Sun GridEngine compute cluster (or a compatible product, such as OGE or OGS). The framework talks to SGE via DRMAA, so you will also need the DRMAA Python bindings. The framework contains a class to talk to the SGE installation available in the Sali lab - `WyntonSGERunner` in `saliweb/python/saliweb/backend/__init__.py`. To work in your environment, add a new `SGERunner` subclass to that file (see the implementation of `WyntonSGERunner` for an example) setting the *SGE_CELL* and *SGE_ROOT* environment variables appropriately, setting *DRMAA_LIBRARY_PATH* to the location of your `libdrmaa.so` file, setting *_runner_name* to a unique value, and setting *_qstat* to the full path to your `qstat` binary.
- MODELLER. The framework needs your academic MODELLER license key (in order for the *saliweb.frontend.check_modeller_key()* function to work). Provide this with the scons *modeller_key* variable when running *scons install*.
- Web server. The frontend consists of Python Flask scripts which need to be hosted by a web server. Apache with mod_wsgi is used in the Sali lab, but other web servers would probably work too (the only assumption made in the code is that files uploaded to the web server end up owned by the *apache* user, but this would be easy to change). The web server does not have to run on the same machine as the framework, but it does need access to the same filesystems (see below).
- MySQL database. Both the frontend and backend need access to a MySQL database in which the jobs are stored. This requires the Python MySQLdb package. The framework itself needs no special access to the database, but each web service that uses the framework needs its own database. (The MySQL server can reside on a different machine to the framework if desired.)
- Unix user. Each web service that uses the framework runs under its own user ID, and the jobs are run on the SGE cluster using the same ID. Thus, generally admin access to the cluster is required to add these users.
- Filesystems. Each web service that uses the framework needs access to a local filesystem on the SGE submit host. This will be used by the web server to deposit newly-submitted jobs (the 'incoming' directory). The filesystem needs to support POSIX ACLs (this generally rules out NFS) since the directory will be owned by a Unix user unique to that web service, but will have a POSIX ACL applied to allow the Apache user to write files into it. Each web service will also need a directory on a shared volume, visible to the cluster nodes, where the files for jobs that run on the cluster are placed (the 'running' directory). (If space is limited on the network storage, web services can also be configured to move the files back to the cheaper local filesystem - the 'completed' directory - when the job is done.)

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

S

Index

T

U

V

W