SPC User's Manual v0.1

SPC, the Scientific Platform for the Cloud, is a platform for easily hosting scientific applications in the cloud. It is described fully in the following paper:

> W Brewer, W Scott, and J Sanford, "An Integrated Cloud Platform for Rapid Interface Generation, Job Scheduling, Monitoring, Plotting, and Case Management of Scientific Applications", Proc. of the International Conference on Cloud Computing Research and Innovation, Singapore, IEEE Press, October 2015, pp. 156-165, doi:10.1109/ICCCRI.2015.24

If using SPC for your research, please cite this paper in your publications.

Getting Started

The latest version of SPC can be downloaded from https://bitbucket.org/whbrewer/spc. This was developed using Python 2.7 for compatibility with other modules, and thus is not currently compatible with Python 3.

Currently, SPC cannot be installed as a typical application but simply runs from wherever the directory is located. To setup SPC, one needs to first generate a database, and also a Python configuration file (src/config.py) as follows:

- `spc init`

To start running the server, then type:

- `spc go`

This will start the web server listening on port 8580. Then, open your web browser to: http://localhost:8580/ (or http://127.0.0.1:8580, or http://0.0.0.0:8580).

"`spc init`" will also install a simple Python demo app called simply "dna", which asks the user to input a string of DNA, and then will analyze its content.

Installing Apps

Adding apps can be accomplished in two ways: (1) interactively via the user interface, or (2) via the command-line options (see section on package management).

To install or modify apps interactively, one must be logged in as user *admin* (default password *admin*). Note, if user authentication is turned, off one will still need to login as admin, by clicking the hamburger icon ≡, logging out, and then logging back in as admin. To install an app interactively, click the *Apps* button, then click the *+Add* button. Installation of an app requires five steps:

1. **Database entry setup** – The first step is to store some basic information about the app into the database:

    - **Name** – A unique name for the app, with no spaces or punctuation marks.
    - **Description** – Description of the app. What does it do?
    - **Input format** – There are three possible options for input format: (1) namelist, (2) INI format, (3) XML format. In the future, support may be added for JSON format as well.
    - **Command** – This is the command to run to start the simulation. This command is being run from the working directory of the case, which is user_data/*user*/*app*/*case* (e.g. user_data/joe/mendel/c82d3f). For example, this may look like:

      ```
      ../../../../apps/mendel/mendel
      ```

      (or `mendel.exe` for a Windows machine). For a Java application, this may look like:

      ```
      /usr/bin/java -jar ../../../../apps/jmendel/dist/jmendel.jar
      ```

2. After setting up the database entry, to finish setting up the application will require several more steps, which can be controlled from the App Edit page, which can be accessed by clicking the cog wheel on the App Edit page ⚙

    a. **Upload input file –** To accomplish this step, click the "Configure inputs" button, and following through the instructions. One must upload an input file that is consistent with the input format specified in step 1. So, for example, if namelist input format is specified, the upload file must be in namelist format. Also, the name of the input file should be the name of the app with the following extension:
        - **namelist.input** → *appname*.in (e.g. mendel.in)
        - **INI format** → *appname*.ini
        - **xml format** → *appname*.xml

    b. **Setup HTML template file –** one of the things that the upload input file format does is to create an HTML template file, *appname*.tpl, in the `views/apps` folder.

    c. **Static assets –** It is also possible to include static assets such as JavaScript files, CSS files, etc. which currently cannot be uploaded through the web interface, but can be added later manually by copying the files to `static/apps/appname`. They can be referenced later in the `views/apps/appname.tpl`, for example refer to the file mendel.js as follows:

    ```
    <script type="text/javascript" src="/static/apps/mendel/mendel.js"></script>
    ```

3. **Upload binary file –** To upload a binary file, click the "Configure Executable" button. Please note that the binary file must be compiled on the same operating system as the host machine running SPC. So, for example, if SPC is running on an Amazon EC2 instance running 32-bit Linux with a certain glibc version, it should be compiled on an equivalent machine running the same glibc version.

4. **Setup plots –** To configure plots, click the "Configure Plots" button, then click "Add Plot". Each plot must have at least one datasource. So, once a plot has been setup, one must click the "datasource" link connected with that plot, and then click the "Add Data Source" button. While there are options for label, plot type, and color, these are all stored together in JSON format under one field in the database called data_def, short for data definition.

Pre-/Post-processing

The code for pre- and post-processing is in the `processing.py` file. This feature can be turned on by setting the pre-processing option in the database setup to the input filename. However, coding to handle the pre-processing step must be manually added to the `processing.py` file.

Pre-processing

The pre-processor is run just before starting the executable in the function `execute()` in `main.py`. This feature is called as:

```
if myapps[app].preprocess:
   run_params,_,_ = myapps[app].read_params(user,cid)
   processed_inputs = process.preprocess(…)
```

This feature can be used for things that need to be changed just before running. Some examples:

- Your program writes its output to a different file than *appname*.out, e.g. out100.00 where 100 is an entry input by the user in the input form. Therefore you can add a couple lines in `execute()` such as:
  ```
  if myapps[app].preprocess == "terra.in":
     myapps[app].outfn = "out"+run_params['casenum']+".00"
  ```

- Your program doesn't actually use an input file, but rather uses command line switches to control certain behavior. Since SPC requires an input file to interact with the user interface, one can use the pre-processing option to convert a file that looks like:

  ```
  [BASIC]
  g_popsize = 100
  o_polytype = NBH
  …
  [MUTATIONS]
  f_del_prop = 0.9
  h_dominance = 0.5
  …
  ```

  to a file containing set of switches that the program reads:

```
-x2 -s2 -n100 -v5 -r10 -k1 -i4 -j0.5 -f0.9 -g100 -oNBH -h0.5 -c5 -u5
```

- This was accomplished by adding the following code to the `preprocess()` function in `process.py`:

```
if fn == 'fpg.in':
   # convert input key/value params to cmd-line style args
   for key, value in (params.iteritems()):
       option = '-' + key.split('_')[0] # extract 1st letter
       buf += option + value + ' '
   sim_dir = os.path.join(base_dir,fn)
   return _write_file(buf,sim_dir)
```

- The pre-processing option may also be used if one needs to write e.g. a PBS run script `pbs.script` file for running parallel applications via MPI.

Post-processing

The post-processor is called when the user clicks on any plot. The post-processor may be used to convert raw output data into JSON form that is needed for a programs like the Flot JavaScript plotting program to plot the files correctly. This function is defined in the file `process.py`. Here is the doc string for the `postprocess()` function:

```
"""return data as an array...
   turn data that looks like this:
   100   0.98299944  200    1.00444448   300    0.95629907
   into something that looks like this:
   [[100, 0.98299944], [200, 1.00444448], [300, 0.95629907], ... ]"""
```

User Authentication

User authentication can be enabled or disabled by setting the `auth` value in `config.py` to either `True` or `False`. There are two default accounts that are setup when the running "spc init". They are: *admin* (password *admin*) and *guest* (password *guest*). The admin user can install, configure apps, run cases, and also can manage other users accounts (presently this is just limited to deleting other accounts, but may in the future include some options such as setting the default priority level, or disk quotas, etc.).

The Web Server

Currently, SPC is setup to use Bottle's built-in web server, which works fine as a development server, but not as a production server. This can easily be changed to use a multi-threaded server such as cherrypy, bjoern, tornado, gae, etc. by changing the server variable in config.py, e.g.

```
server = 'cherrypy'
```

Of course, this assumes that cherrypy has already been installed (e.g. sudo pip install cherrypy).

Package Management

Because it may take some time to setup an app, especially the plots, it is possible to save the configuration as a JSON manifest file called `spc.json`. Therefore, it is possible to create an SPC package of an app, which is simply a zipped archive containing the input file, binary, spc.json manifest file, and then make it available on the Internet (e.g. by hosting it on github.org or bitbucket.org). In this way, anyone else running SPC can easily install your app by running the command:

```
spc install http://url/to/package.zip
```

or by running the command on a file:

```
spc install /path/to/package.zip
```

For Mac OS X users, an example can be installed by running the following command:

```
spc install https://github.com/whbrewer/fmendel-spc-
                osx/archive/master.zip
```

The Job Scheduler

SPC currently has two job schedulers available: (1) the uni-job scheduler, and (2) the multi-processing scheduler. The uni-job scheduler will only schedule one job at a time. It is a simple, but robust implementation that works well for user's running simulations on their laptop for example. The multi-processing scheduler uses Python's multiprocessing module and includes synchronization primitives such as `Lock` (mutex) and a `BoundedSemaphore` for ensuring that only a user-specified number of jobs can run concurrently. To change scheduler options, modify config.py.

To set the uni-job scheduler:

- `sched = 'uni'`

To set the multi-processing scheduler:

- `sched = 'mp'`
- `np = 2`

np is short for number of processors, but really represents how many processors can be scheduled concurrently. For example, if the value is 2, two jobs that require a single processor can be scheduled, or a single job that requires 2 processors can be scheduled.

Setting up MPI-based applications

- Install MPI software (e.g. www.mpich.org)
- In config.py, set the mpirun parameter, e.g.

```
mpirun = '/usr/local/bin/mpiexec'
```

- In `config.py`, the `np` parameter has to be set to a value of greater than 1.
- After clicking Start, then Continue, select the number of processors to use for this run:

- For example, setting the value to 2, will execute the command:

```
mpiexec –n 2 app.exe
```

Miscellaneous

- How to pass a hidden "case_id" value used in the interface into the file?
    - Create a variable called "case_id" in the input file (e.g. case.ini). When you are uploading and parsing the input file and are asked to assign html entity types for each parameter, set the type to "hidden".

Known Issues

- HTML input elements, which are disabled will cause problems. In essence, they will be interpreted as checkboxes that are not checked, so the values will be set to `F`, which may cause problems when the simulation program tries to read the value and is expecting say an integer value. Therefore, in lieu of using `disabled=true` on input tags, set `readOnly=true`.
- The restart option does not work for apps that use XML input files. This is a problem with the structure that is being output from SPC and needs to be fixed in the future.