Operating Systems 2: Final Project

Jorge Jair Ramirez Estrada

December 2, 2014

1 Synopsis

It is common among biology specialists to use taxonomical trees to illustrate the evolutionary relations among different species. Yet, producing these visual elements is time consuming and usually implies simple repetitive task. If the users could spend less time drawing trees manually, they could spend more time on non-trivial tasks.

The Bath Project aims to give a solution to this problem. It is a web application that receives a tree in *Newick* and produces a graphical representation of the phylogenetic tree with illustrations (when available) for each taxon.

2 Specification and Schedule

We intend to provide a solution for the user requirements by developing the prototype of a web application. The prototype was designed as a simple and portable solution to the problem. It will use bioinformatics libraries (*biopython* and *ETE2*) images available as resources in public databases (*Phylopic* and *Encyclopedia of Life*) to automate the process of generating phylogenetic trees.

In order to reduce the number of requests made to the public databases and to improve the responsiveness of the application we also include a small database to keep cache data. Because querying the public databases implies being subject to latency due to IO access and network speed we designed the core of the public database interface as a multi-threaded process. We also designed the interface with each database's *API* as a data plug-in, so other databases can be easily included in future development.

In order to produce the best possible results, the user should be able to configure options used to generate the phylogenetic tree. The options should allow to select which data source(s) will the application use for the taxa images.

To simplify further maintenance of the application, the language and frameworks used were selected because they are well known. We also decide to make the code openly available to allow new developers to extend the existing code. The project was built in such a way as to make it as portable as possible and manage it's dependencies in an elegant an simple form. A wrapper for everything inside the project (from dependencies and libraries to the source code) was considered to be highly desirable.

2.1 Schedule

2.1.1 Week 1

Work plan for the week:

- Input data: There will be a page with a text area and a file input (for big trees) allowing the user to input a Phylogenetic tree in *Newick* format
- Fetch species data from databases: Using the information from the user, there will be a data plug-in which will request the images from the *Phylopic* web site. The *Newick* tree structure will not be parsed or verified at this moment. A species list can be obtained from the *Newick* raw string
- Show users data fetched from databases: The program will only pick one image for every species the user provided as input and that will be shown in another view
- Identify problems between the provided tree structure and the data in databases: If the species is not found due to spelling mistake or missing images for the species, the user will be notified. The species will be used for the visual tree generation but will have no image.
- Identify problems with images missing for all species: If there is no data for a species in the data source, it will let the user know

Real work done during the week:

- Input data: Specified above
- Fetch species data from databases: Specified above, only with the addition of the Encyclopedia of Life connection also established
- Show users data fetched from databases: Specified above
- Identify problems with images missing for all species: Specified above
- Select databases: Specified in Week 2

2.1.2 Week 2

Work plan for the week:

- Data sources as general abstract class: Each data source should inherit from the same general abstract class so that they can be interchangeable
- Select databases: There must be UI elements to chose from different databases and the back-end logic to query said databases.

Real work done during the week:

- Data sources as general abstract class: Specified above
- Request data sources for multiple images of the same species and let the user chose the ones she likes the most: Part of the work item "Allow user to select images" specified on week 4

2.1.3 Week 3

Work plan for the week:

- Show tree: Build tree with the default images found in the databases and show it to the user.

Real work done during the week:

- Identify problems between the provided tree structure and the data in databases: Specified in week 1
- Cache results: Specified in week 5
- Parallelisation of queries: Not explicitly specified. Added to improve performance

2.1.4 Week 4

Work plan for the week

- Allow user to select images: Build tree with the images selected by the user

Real work done during the week:

- Investigate how to use and integrate Docker in the project: There is a dependency, *ETE2* (the library used to build the trees) which need a very specific configuration on a linux-based system. After investigating a series of alternatives, the team decided to use Docker to manage those dependencies, and thus the technology was added to the solution

2.1.5 Week 5

Work plan done during the week:

- Register queries: Use *SQLite* to register the requests to the server
- Cache responses: Increase responsiveness of the application, using cache on the responses

Real work done during the week:

- Docker image configuration: There were configuration requirements in the Docker image that needed be met to make the application work
- *ETE2* troubleshooting: *ETE2* required extensive work to make it function correctly, since it needed graphic services which were usually unavailable in linux headless server configurations

2.1.6 Week 6

Work plan done during the week

- Testing: Make regression, functional and failure tolerance testing

Real work done during the week:

- Testing: Specified above
- Documentation: This document

3 Functional Attributes

3.1 Architecture

In general terms, this is was designed as a *MVC* application which makes asynchronous *REST API* calls and processes the results to make customizable graphical representations of phylogenetic trees. The solution is encapsulated in light representations of virtual machines.

Going to the specifics:

The development of the system makes use of Docker images, since all the dependencies are managed through it and it simplifies the deployment of the module. The project was developed with a Python web development framework *Django* and uses *SQLite* as the back-end database manager. The architecture of the project is straight forward and uses abstraction to simplify further extension. The only major configuration is which data source to use, both data sources need to be queried in *API* specific ways, this is done by objects that inherit from Data_plugin. Each of the specific implementations of data plugin query a different data source, which makes them easily interchangeable and manageable.

3.2 Persistence

For data persistence, the first question that comes to mind is "Relational or Non-relational?" In this particular instance we saw fit to use relational databases since the data we would be highly structured and we would need to scale vertically. Non-relational databases do not scale well horizontally. The application requires all *CRUD* operations, a relational database will offer the best performance considering the circumstances.

For the first release of the project we will use *SQLite* for practical purposes. It is a simple relational database which can be accessed with the standard

Python libraries. But because we rely on *Django ORM*, this dependency can be easily switched to other relational database manager at a later stage. We use *Django Model* as the abstraction layer to access the database.

The data model used for this application is straight forward. We save each request in a table register as a register containing a unique identifier, the *Newick* representation of the requested tree (regardless the status of the request), data source selected and a timestamp. Given the user only makes an input, it is easier to record said input this way, it doesn't need to be more normalised.

3.3 Tests

The file /bathsite/tests/tests.py has 3 types of tests: ping, destructive and regression tests. Each one of them tests different things and have different objectives

Ping tests: they test the availability the different views within the application. These tests just make sure the views' response is a *HTTP status code* is 200. These are light-weight tests and should be the first group of tests run, since if these do not pass, it would be unlikelly for the others to pass and can also pinpoint the failling view.

Destructive tests: they try to feed the application erroneous input to get the error message. These tests work on the application's input validation. Errors on these tests most likely do not threat the application, it would just mean that a small group of careless users could be getting errors in their responses.

Regression tests: as the application grows it is of prime importance to make sure that new features don't brake previous ones. If any of these tests brake, it would mean there is a serious error, meaning a lot of users might be getting errors to their requests. When a version of the app doesn't pass these tests, that version cannot be put into the *master* branch, and if the version is in *stagging* (if such branch exists) it should go back into *developing*

4 Characteristics as a Reactive Service

4.1 Responsive

The main advantage in the application's design is that it is a stateless request-response application. The user inputs the tree in *Newick* format, chooses a data source and gets back an image. If at a point in time the application is unresponsive the user can just restart the request and have another opportunity to have it processed.

Having the user retrying a request will probably be the last resort. Given the most expensive processing operation is the *REST* calls to the databases, we follow two strategies to make them less costly:

- Reduce the number of requests: We have implemented cache for the requests made to the application. If the request is cached we can build the response tree with the images found in the cache avoiding the *REST* calls

- Make parallel requests: Since each tree could involve a large number of calls, by doing them in parallel application becomes more responsive by potentially an order of magnitude or two

4.2 Resilient

There are several errors that could happen during a request, I will describe the three main concerns we had while designing error handling:

- Incorrect *Newick* tree format: the user can send an incorrectly formatted *Newick* tree which would result in an error when the image creation is requested. This is verified early on when the request reaches the server and if the tree is not correctly formatted the user is notified. We rely on *Biopython* to check the validity of the threes submitted

- Errors during the database's *API* calls: every web call is encapsulated with a try-catch block. If a web call fails 3 times, the user is notified the data source server might be down and to try again later on

- *JSON* serialisation: every *JSON* serialization is enclosed in a try-catch block. If the serialisation fails the user will be reported to contact an administrator because the web response of the *API*s changed. This error will obviously not go away even if the user tries again later

4.3 Elastic

This section is not implemented but given the idea that Docker can start up instances of the service, then a monitoring and scheduler layer could be placed on top of the instances to share the workload among the available instances. Through the logs emitted by the docker images, the scheduler would be able to know which virtual server is best to work on the new request.

Given shortage of resources, fewer instances could be started, the scheduler would need to be intelligent enough to know when to drop a request instead of knocking down a virtual server. Evidently, it is best to leave dropping the request as a last resource. When this happens, the scheduler itself should log the details of the incident so the administrator of the service would know how much more scaling does the system need (easily done with cloud infrastructures).

4.4 Message Driven

Not considered

5 Attributes from *12 Factor App*

5.1 Code base

The code base is in:

https://github.com/dasmesser/Bath

This repository contains (at least) 2 branches: *master* and *developing*. The *master* branch is inteded to contain only stable (though incomplete) versions of the application. As a last resort, where everything starts going wrong in the development process, you can always go back to the last commit in the *master* branch and restart the last sprint. As for the *developing* branch, it is used to store the unstable versions of the application. If there is a major implementation of a feature, it is recommended to use yet another branch forking from *developing*, as happened with a branch called *ete2Tests*. If the project starts to integrate more than 2 developers, it would be convenient each one of them has it's own branch forking from *developmen*t or from a specific massive feature which has it's own branch. This way the programmer may work on her own untainted branch before merging it to the more public *developing* branch or the feature branch. If the number of programmers grows, it is also adviced to make a *stagging* branch to 'between' the *master* and *developing* branches. So whenever a developing sprint is done and testing is due, the *developing* branch can be merged to the *stagging* branch and be tested. If tests are successful then it is merged to *master* if they are not it is merged to *developing* for bug fixing.

5.2 Dependency Management

Given the libraries we use in the web application, we require a Linux-based environment for *ete2* library, it was decided to use *Docker* to manage dependencies. *Docker* is a program to emulate light virtual machines, which can be easily discarted and recreated. *Docker* provides the functionality to create *Docker images* which are the instructions to reproduce the *Docker* environment. This allows to manage OS level dependencies and library dependencies. For an environment to run Bath it only needs the to have *Docker* installed and the *Docker image*. These images can be further modified and commited to the image repositories if necessary. To this point Jair Ramirez and Abdul Cordoba have been successful running Bath with Windows and Macintosh machines respectably.

5.3 Configuration Schema

There are several configurations that can be managed by scripts when starting the application, where to make the git repository, what branch to pull and executehow to map the ports (done automatically by the docker image, but manually configurable).

A command along the following lines would need to be implemented:

sudo docker run -t -i -p 8000:8000 abdulcordoba/bath /bin/sh -c ”Xvfb :0 & export DISPLAY=:0; cd home; git init; git pull https://github.com/dasmesser /Bath.git ete2Tests; cd bathsite; python manage.py runserver 0.0.0.0:8000”

Here we can appreciate the -p para that maps the local machin’s port 8000 to the *Docker image’s* 8000 port. Next, inside the argument passed to /bin/sh we find the folder being initialized as git repository is ’home’ and that it is pulling from user dasmesser’s repository called Bath, hosted in github, branch ete2Tests. When running *Django*, it is configured to accept requests from any ip (not just the local one as with the default configuration), going to the 8000 port.

All this configuration settings can be altered without changing the code base.

5.4 Support Services

So far we can declare to use two supporting services to get the images for the species requested by the user, the *Encyclopedia of Life API* (http://eol.org/api), and the *Phylopic API* (http://phylopic.org/api). The use of both are explained here.

Encyclopedia of Life (EoL): When the app receives the species to be searched for, the first thing to do is to look for the ID of the said species in EoL, achieved by calling *REST API*:

http://eol.org/api/search/1.0.json?q=[species]&page=1&exact=true

Where [species] must be replaced by the name of the species the application is looking for. EoL specifies that if there are more than 30 matches, the results will be splitted in several pages, we are only interested in the first matches (we do not expect that many results to come back any way) , so we specify ”pages=1” in the url. We also set the argument ”exact” to true to only get the result that match perfectly with our query.

For example, for the call:

http://eol.org/api/search/1.0.json?q=Ursus&page=1&exact=true

Corresponds the following json:

{
”totalResults”: 1,
”startIndex”: 1,
”itemsPerPage”: 30,
”results”: [
{
”id”: 14349,

"title": "Ursus",
"link": "http://eol.org/14349?action=overview&controller=taxa",
"content": "Ursus Linnaeus, 1758; Ursus; Ursus Arctos Bruinosus; Ursus Arctos Ssp."
}
],
"first": "http://eol.org/api/search/Ursus.json?page=1",
"self": "http://eol.org/api/search/Ursus.json?page=1",
"last": "http://eol.org/api/search/Ursus.json?page=1"
}

On this instance we are interested in the dictionaries' 'id' fields inside the 'results' JSONArray. Those are the ids of the species the user is looking for. The following *REST API* we call is:

http://eol.org/api/pages/1.0/[species_id].json?images=10&videos=0&
sounds=0&maps=0&text=0&iucn=false&subjects=overview&licenses=all&
details=false&common_names=

Where [species_id] is the id of the species (which is gotten from the previous step) The json for this call is considerably larger (although we specify we only want the images, a lot of data is also received). Yet on the json there is a JSONArray matched with the key "dataObjects", each index contains a dictionary with a key called "dataObjectVersionID" which is the ID for the image. EoL identifies each sound, map, video, image, etc. as a dataObject, thus it is needed to find the id for the specific data object we are looking for (done in this step), and then its url. This can be achieved through the *REST API* call:

http://eol.org/api/data_objects/1.0/[object_id].json

Where [object_id] is the idof the dataObject from the previous step. The resulting json is also a little lengthy, but it has an JSONArray mapped with the key "dataObjects" inside the first index (since we are only performing a query for one object_id) contains the url of the image, mapped with the key 'mediaURL'

Further documentation in http://eol.org/api

Phylopic: The first *REST API* needed to call has the purpose of knowing the id Phylopic gives to the species being queried. The url is:

http://phylopic.org/api/a/name/search?text=[species]&options=illustrated

Where [species] is the name of the species whose id we need to find. The 'options' parameter indicates the caller is interested in knowing which results are illustrated and which are not. The json has an JSONArray mapped to the key "result", each index has an id for the species (it is not unusual in Phylopic for some species, such as the Homo-sapiens, to have multiple ids) and each id will declare if it is illustrated, the application searches for the ones that are. The application needs to get the images related to the newly found id:

http://phylopic.org/api/a/name/[species_id]/images?options=pngFiles

Where [species_id] is the id of the species the user is interested in. Inside the resulting JSONthere is a dictionary mapped to the key 'result', then there is JSONArray mapped to 'same' with only one index (because in the url we only requested the pngFiles for the id), inside that index there is an array with several dictionaries specifying the url for the image in different sizes.

Further documentation in http://phylopic.org/api/

5.5 Develop, Release, Run

The main difference between the dev and prod environments is the server running the apps. Django is a development server and (according to it's documentation) should not be used in production environments. The script given in the "Configuration Schema" section can be used to run the server in development mode.

To make it run in a production environment it is recomended to use WSGI. The WSGI setting-up and configuration is easy. It is best summarized in this document:
https://docs.djangoproject.com/en/dev/howto/deployment/wsgi/uwsgi/

The only modifications needed that differ from the tutorial would be:

- chdir: would need to be configured to the root of te project
- socket: assign to desired port
- processes: adjust to CPU capacities of th machine
- daemonize: point to a desired log directory inside the solution

5.6 Process Model

There are several parts of the project that can be easily deattached and the project would continue working:

Database: currently we are using an *SQLite* implementation for data persistance, yet it is possible to replace this instance with another database manager such as *MySQL*, given that *MySQL* instance is correctly configured (the tables exists, the authentication doesn't block the connection, etc.).

Data Pluggins: there are two data pluggins as of the creation of this document, one for Phylopic API and another for Encyclopedia of Life API. Yet any of these pluggins could be removed at runtime by removing the correct variable from the /bathsite/utils/constants.py file and the creation instruction in /bathsite/data_pluggins/pluggin_factory.py

Templates and static pages: given the fact that Django can recompile modified files even while it is running, it is possible to edit those files on-the-go.

These kinds of hot-swaps in the project are highly unadvisable in a production environment, they should be tested throughly in a stagging environment before attempting them in production.

5.7 Port Management

Given *Docker* feature to map image ports to the local machine's port, port management becomes easy. Nontheless, one should be careful when assigning ports, it is adviced against overriding well-known ports, as 80, 25, 21, etc. that the local machine may be using for common services. Generally, depending on the other services the machine may be using, registered-ports are a good option to map the internal ports to, these ports range from numbers 1024 to 65535.

One must also be careful, since the default port for Django is 8000, while wsgi uses 49152. So if the user maps the image's port 8000 but starts up a wsgi service, external communications will not go through.

5.8 Concurrency

Not considered

5.9 Disposable

This was not implemented as part of the current version of the project, yet as a stateless internet application, it would fit nicely. Probably the most effective solution would be the one described in the ”Elastic” section, near the begining of the document. Said solution would include several instances of the *Docker image* application running on the same machine, they would all be able to communicate to a scheduler which would receive the requests and forward them to the most under-loaded instance of the application at the time.

Given each instance is quickly disposed of and quick to set-up, the scheduler could be programmed to monitor the load on it and know when to shut down processes and when to start them again when they are more needed. To communicate different *Docker images*, the following tutorial is recommended:

https://docs.docker.com/userguide/dockerlinks/

5.10 Dev/Prod Relationship

This was not implemented as part of the current version of the project. There are no tests about creating dev/prod environments, yet it can be done with the *Docker image* initialization scripts in the ”Configuration Schema” section and the ”Develop, Release, Run” section. The largest and most significant change when creating a development or production environment are whether to use Django or WSGI as container for the application.

It will be noted as a work item for future developers to automatize the creation of these environments and test they run according to their specific requirements.

5.11 Log schema

A Log schema is not implemented in the project's current version, yet it should be considered for future work. There are several places where logs would be useful:

- On start: the log should contain when has the application started
- Errors: whenever something fails (access to the database, json parsing, *REST API* calls, etc)
- Cache or Web: the system should record how many calls were retreived from the cache (thus being way faster) and how many had to be gotten from the web. This could help determine if the cache algorythm is useful or if it needs more space to work properly
- CPU and RAM usage: with the *psutil* library for python (reference in Bibliography section), the developer can get such data and log it periodically during the day to know spikes in usage and device more effective strategies to meet demand.

There are plenty of options for logging libraries, one of which is the logging module included in most python distributions. Here is a link to a tutorial:

https://docs.python.org/2.6/library/logging.html

5.12 Administrative processes

Not considered

6 Conclusions

There were several key learning items I could acquire with this project:

1) Source Control: When working in teams (even in small ones) it is fundamental to have some sort of source control tool and everyone should know how to use it. Even if it is only 1 man project, having the ability to branch, make commits and go back to those commits is of paramount importance. To this point I hadn't had the need to force myself to learn this valuable skills. In this

project I always had at least 2 branches, the developing branch and the master branch which I constantly merged and branched. I got a lot of practice on those, also I explored gitignore files, and pushing and pulling code from several computing instances.

2) Dependency Management: In the "School World" where you program something once to be run once on the professor's computer whose specifications and dependencies installed are perfectly known, there is little need to do dependency management. Yet, on the Real World where you don't really know where will the code end up running, dependency management is critical. I started up using virtualenv to manage dependencies, yet at some point it was not enough and I had to use Docker to emulate a whole operating system. It was a learning experience I had not come in contact with and one that I ended up enjoying and appreciating a lot.

3) Asynchronus *REST API* calls: Web calls are expensive time-wise and in the meantime the processor doesn't do much, so the computer resources are wasted. To avoid that as much as possible it is imperative that a web application always uses asynchronus *REST API* calls. In the past I had tried doing this in another project with C but failed. Multithreading and resource management is a critical skill to have, yet a difficult one to achive given the complex nature of multithreaded algorythms. Luckly I could witness the development of one such algorythm in python.

4) Python and *LaTeX*: Although no programmer can be expected to know every single programming language (there ought to be hundreds, or even thousands of them), every programmer should know a compiled language, a web development language, an interpreted language and so on. I hadn't had the experience of working with an interpreted language such as Python and I completely ignored such languages as *LaTeX* existed. It has been enriching experience working with them.

7 Referencias

Usar bibtex. He aquï¿$\frac{1}{2}$ por ejemplo cï¿$\frac{1}{2}$mo hacer referencia al tutorial de Django [1]. http://pythonhosted.org/ete2/install/index.html https://pypi.python.org/pypi/psutil http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

References

[1] Writing your first django app, part 1. `https://docs.djangoproject.com/en/dev/intro/tutorial01/`. Visto: 2014-10-14.