Operating Systems 2: Final Project

Jorge Jair Ramirez Estrada

November 28, 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 must be able to configure the tree and what data source will the application use as source for the images.

Since people who will maintain the web application will most likely not be software developers, the language used and frameworks most be something they already know, in this case Python and the code must be in an open repository. Also the project must be 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) would 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

Estrategia general de pruebas a implementar. Notese que son pruebas automatizadas.

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

5 Atributos de *12 Factor App*

5.1 Base de cï¿½digo

5.2 Manejo de dependencias

5.3 Esquema de configuraciï¿½n

5.4 Servicios de apoyo

5.5 Construye, libera, corre

5.6 Modelo de procesos

5.7 Manejo de puertos

5.8 Concurrencia

5.9 Descartable

5.10 Relaciï¿½n dev/prod

5.11 Esquema de bitï¿½coras

5.12 Procesos administrativos

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 web 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 web 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¿½ por ejemplo cï¿½mo hacer referencia al tutorial de Django [1].

References

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