Operating Systems 2: Final Project

Jorge Jair Ramirez Estrada

November 26, 2014

1 Synopsis

It is common among biology specialists to use taxonomical trees to illustrate the evolutionary relations among different species. Yet, this visual elements take a considerable amount of time to draw and are not something that requires a high level of expertice. If the user could spend less time drawing trees, she 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 format of the species the user wishes to include in the taxonomical tree and a source from which the images should be retrieved. The output of the web application is the graphical representation of the tree with illustrations of the species.

2 Specification and Schedule

To cover the need specified above, the easiest, more portable (from the user's perspective) and straight forward solution would be a web application that queries different data sources (such as Phylopic or Encyclopedia of Life) to get the images for the species requested.

The solution would need to have a database to have persistent registers on the requests made to the service. To increase responsiveness the solution must implement result caching and multithreaded web api calls to the data sources. 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 very 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 pluggin that will request the images from the http://phylopic.org/ site. The Newick tree structure wonÂ´t be parsed or verified at this moment. Species list can be obtained from the Newick 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 inexistance of 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 in 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 interchangable
- Select databases: There must be UI elements to chose from differet databases and the back-end logic to query said databases.

Real work fone in 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 in the week:

- Identify problems between the provided tree structure and the data in databases: Specified in week 1
- Cache results: Specified in week 5
- Parallelization 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 in the week:

- Investigate how to use and integrate Docker in the project: There is a dependency, ete2 (the library used to consctruct the trees) which need a very specific configuration on a linux-based system. After investigating a seies 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 in 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 in the week:

- Docker image configuration: There were configuration requirements in the Docker image that needed be met to make the application work
- Ete2 troubleshooting: Had a fair amount of trouble making ete2 work correctly, since it needed graphic services which were usually unavailable in linux server implementations

2.1.6 Week 6

Work plan done in the week

- Testing: Make regression, functional and failure tolerance testing

Real work done in the week:

- Testing: Specified above
- Documentation: Ahem...this document.

3 Functional Attributes

3.1 Architecture

Speaking in general terms, Bath is an MVC application that makes asynchronus web API calls and process the results to make customizable graphical representations of taxonomical 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. The project was developed with Python web development framework Django and uses SQLite as a database manager. The architecture within the project is preaty straight forward. The only major configuration is which data source to use, both data sources need to be queried in different forms, this is done by objects that inherit from Data_pluggin. Each of the specific implementations of data pluggins query a different data source, which makes them easily interchangable and managable.

3.2 Persistance

For data persistance, 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.

For the first release of the project we will use SQLite for practicity purposes. It is an SQL database that comes along and integrated within Django. Most probably later on, the data will be migrated to another manager like MySQL, which has better support and documentation.

The data model used for this application is preatty straight forward. We save each request in a table register as an ID, the Newick representation of the requested tree, whether the request was a succes or a failure, data source selected and date made. Given the user only makes an input, it is easier to record said input this way, it doesn't need to be more normalized, since that would make the data model more complex, and insertions and queries more complicated,given the join operations implicated.

3.3 Tests

Estrategia general de pruebas a implementar. Nï¿½tese que son pruebas automatizadas.

4 Characteristics as a Reactive Service

4.1 Responsive

The main advantage in Bath's design is that it is just, what I would call, a request-response application. The user sends the tree in Newick format and chooses a data source and gets back an image. If at a point in time the application is oveloaded the user can just refresh it's requests and have another shot to have it processed.

Having the user refresh its request is the last resource we would like to rely upon. Given the most expensive operation done is web API calls, we follow two strategies to make them less costly:

- Don't make them: 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 web calls
- Make them parallelizable: Since each tree could involve anywhere from 5 to docens of calls, by parallelizing them the application becomes more responsive by potentially an order of magnitude or two.

4.2 Resilient

There are several errors that could happen in the application, I will describe three:

- Incorrect Newick tree format: the user can send an incorrectly formated Newick tree which would result in an error when the image creation is requested. This is verified early on when the request reachs the server and if the tree is not correctly formated the user is notified.
- Web call failure: every web call is encapsuled 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 Serialization: every json serialization also has a try-catch associated with it. If the serialization fails the user will be reported to contact an administrator because the web response of the APIs changed. This error will obviously not go away 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 emited by the docker images, the scheduler would be able to know which virtualized server is best to work on the newly come request.

Given shortage of resources, fewer instances could be started, the scheduler would need to be inteligent enough to know when to drop a request instead of knocking down a virtualized server. Evidently, it is best to leave droping 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 configuracïï¿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 Relacïï¿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.