QAWeb - Using Django via SSH for QA Statistics

by

Spyridon Antakis
spyros@loonydesk.com

August 18, 2012

Abstract

In every software development cycle, the Quality Assurance (QA) constitutes probably the most vital phase for the success of the deliverables. Thus, the existence of an automated tool for reporting the test case results on a daily basis becomes a necessary part of that process. This paper describes a solution for building your own flexible web interface for reporting QA-Statistics, named QAWeb. In practice, it presents how you could use the Django web framework in order to implement a simplified, but nevertheless flexible and sufficiently powerful QA web tool.

1 Introduction

Nowadays, it is very common every corporation to have it's own custom internal tools for enhancing their *Software Development Lifecycle*. However, most of the time these tools have been built with a certain philosophy in mind. Their architecture is very hard to change and sometimes even obsoleted. A tool that was meant to solve a certain problem in time does not mean that will satisfy successfully any upcoming challenge. The needs and requirements among different teams and projects significantly varies and re-usage of existing tools could potentially become a productivity bottleneck. Although there are no standard guidelines for deciding whether or not it is better to re-use an internal tool, there are certainly some basic and fundamental prerequisites, such as *scalability*, *low complexity*, *usability* and *flexibility*. If a tool fails to provide any of these features for the purpose of its use, then its adoption **must** be reconsidered.

One of the most important aspects of a complete software development process is the Quality Assurance (QA) of that software. Various test frameworks and a number of different *programming* or *scripting* languages are creating a multi-dimensional testbed for QA engineers. In this customised environment, a tester will perform automated but also manually software testing on a daily basis. As a result, there is a need for capturing the results of each performed test *real-time* and report meaningful statistics on an *ongoing way*. Having all the above in mind, a web interface gets more attractive as a potential candidate for supporting this feature. Setting as only requirement a peaceful integration with the technologies that QA engineers are using.

This paper proposes the *Django* framework as a way to build a lightweight web tool for reporting *QA-Statistics*. It is considered a relatively easy approach, that could give to a company the desired functionality. In section 2, we briefly discuss the essentials of *Django* and we argue for choosing it as our web framework. Section 3 presents the proposed architecture for *QAWeb* and reveals the rationale behind it. In section 4 we summarise our thoughts and we conclude for the advantages and disadvantages of our design. Finally, at the same section we provide a direction for possible future work and we give ideas for extending *QAWeb*.

2 Django in a nutshell

"Django - The Web framework for perfectionists with deadlines."

Django is a **high-level** python web framework that encourages rapid development and a clean, pragmatic design [1]. Architecturally it is structured from three core layers, namely the *Model*, the *View*, and the *Template*. By developing based on these layers, you gain a valuable abstraction without compromising necessarily on programming power. The fact that Django is written for python makes the entire framework extremely adjustable. In practice you enjoy all the goodies that the *Python* language provides, but at the same time, you are offered a great interface to present your web content as you desire.

At this point someone could argue that there are so many python web frameworks that can be used, so why use Django. Again it comes back to the company's needs, developer's preference and if what you are trying to achieve is sufficiently covered by the framework's capabilities. In the case of QAWeb the admin panel that Django has *out-of-the-box*, its excellent online documentation and the truth behind the quote: *"...for perfectionists with deadlines"*, constitute the main reasons of that choice. It is incredible easy to start coding and implement your own models, when you finally understand the logic behind the Django framework. That does not mean that there are no *pitfalls* or *limitations*. However, compared to the benefits obtained for building QAWeb these are just minor for the goal of this project.

In this document we avoid to dive deeper into the *Django World*, as it is out of the scope of our purpose. We consider that the reader is either already familiar with the framework or he does not have any previous background. In either case, we ensure that it would not impact the comprehension of this paper, since we fully clarify all the parts which are Django related. Although, we definitely encourage the readers to try Django and visit its official webpage [1], as it would open more their thinking about web frameworks.

3 QAWeb Architecture

There are *4 essential parts* on which the QAWeb architecture is focused:

1. Database Structure & Models
2. Control over the data
3. Presentation & Reporting
4. Simplified Automation

As we will see, Django successfully contributes to most of these essentials and makes the design much easier without significant effort. All the parts are combined aiming to deliver a tool that will offer: *Scalability*, *Portability*, *Flexibility*, *Usability* and *Accessibility*. As we continue with the analysis, you will be able to easily understand where these assets comes into the picture.

3.1 Database Structure & Models

Django automatically associates the defined models with the structure needed on the used database backend (e.g. MySQL). That means that when you finish writing all of your models, there is a *"Django way"* to auto-generate the needed tables in the database via some utility scripts provided by the framework. A *model* is nothing more than a *python class*, which can

contain any field type provided by the Django framework, such as *integers*, *urls*, *boolean*, *textarea*, *characters*. Of course, the user is always able to create its *own* field types if needed, by using *powerful relations* among models and normal python *class inheritance*. The best way to explain the rationale behind this philosophy is to walk the reader through an example of the python code that is used by QAWeb.

```
from django.db import models

class Report(models.Model):
        title = models.CharField(max_length=100)
        description = models.TextField()
        configuration = models.TextField()
        project = models.ForeignKey(Project)
        team = models.ForeignKey(Team)

class Team(models.Model):
        team = models.CharField(max_length=100)

class Project(models.Model):
        project = models.CharField(max_length=100)
```

In the above *source code* there are 3 defined models: *Report*, *Team* and *Project*. As you can notice the Report model contains both the *Team* and *Project* models as **foreign key relations**, which in practice means that each created *Report* instance will contain a project and a team field type. That's a very basic example of an abstract re-presentation of a model structure. The key point is that we only need to configure a database, a user for that database and then run an utility script to create all the necessary tables. That's a very *flexible* feature that Django provides and it really makes development way easier. Of course, that does not come without a *trade-off*. Any potential future changes to the model structure will have to result to a database structure update. Sometimes that's pretty straightforward and easy to apply. However, if the model changes are significant and stored data size is big, then that could become a more complex problem to solve.

3.2 Control over the data

Control over the data refers to *2 different aspects* of the system. The first part is the control that an administrator has over the data stored in the database and the second is the control features that a user enjoys via the front-end web interface. As we already briefly mentioned, one of the most powerful things that Django provides is the pre-build Admin Panel. Through that interface an administrator can easily `edit/delete/create/` any of the system models, in a few seconds. If you add also the fact that the admin panel could be fully customized to include more functionality, then you come up with a reliable and flexible way to control your environment with minimum effort. On the other hand, when you look the control that you could have at the user's/visitor's level, the Django/Python capabilities continue to be impressive. The framework supports application modularization, which in practice means that any open source django-apps can be integrated into our system and offer very useful features [2]. Moreover, any python library can be imported as normal, exposing the framework's feature development at the python levels. In our case, we were looking to create a very basic functional environment and we did not focus on using existing apps. After all QAWeb is just a new idea for getting QA-Statistics via script calls and it is more than certain that additional changes would be need for different companies. We took advantage mostly of what the framework has included into

his internal API over the years, with respect to apps and we only introduced the search engine [3] [4] and the generation of graphs [5], as installation requirements.

3.3 Presentation & Reporting

The approach for presenting our data was already defined for us. Django uses a *Model*, *Template*, *View* (MTV) variation that resembles a lot the *Model*, *View*, *Controller* (MVC) philosophy. Both MTV and MVC are types of computer user interfaces that separate the representation of information from the user's interaction with it. Therefore, the QAWeb data presentation is completely separated from the core functionality of the platform. The *CSS/HTML* templates for QAWeb were written from scratch and their main goal was to satisfy the need for a simple, clean and comprehensible reporting point for QA statistics via web. The graphs were chosen to be pies, but **matplotlib** supports several different modes and types that can be used. The intention was not to *over-complicate* the design, but only demonstrate how it is possible to generate and update dynamically the graphs based on the content of the QAWeb database. Additional features such as the auto-completion for search and the RSS feeds were added, in order to improve the total experience for the visitors. The idea is that any further development on QAWeb will not affect the presentation and even if it does, the changes that will be required, will be minimum. That's another major advantage that comes out of the usage of Django.

3.4 Simplified Automation

Automation is probably the most critical part for the daily job of a QA Tester. Our first and only goal was to provide a way that would be sufficient and powerful enough to be integrated inside the scripts for tests. The goal was to be able retrieve the *output information* of each test case after a single run and report it back to QAWeb. On the report of all changes the database would be updated and the web interface will automatically reflect the new state of the graphs.

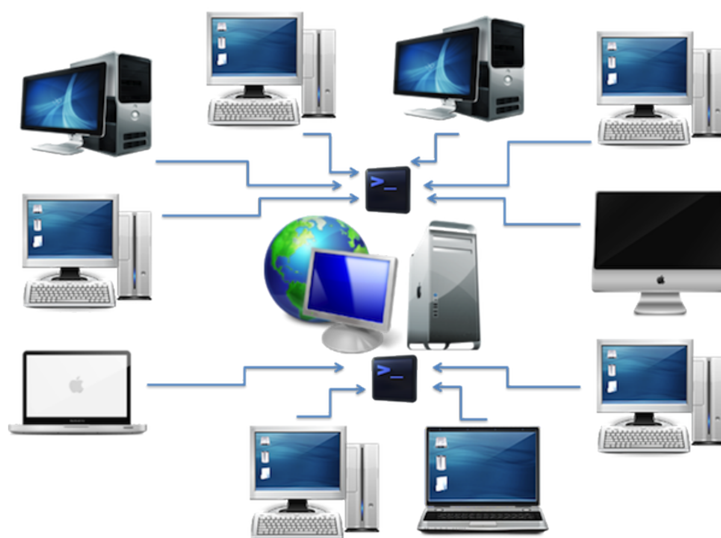

Figure 1: The QAWeb Architecture

By considering that SSH is the easiest way for a QA Tester to reach out from a test case, we designed an architecture on which a SSH connection with a remote command call would be sufficient to update the QAWeb data. In this way, a central server hosts the QAWeb platform and all the devices in the same network can connect via SSH (figure 1) and call a dedicated *"python*

script" with parameter that will update a test case based on a unique ID number. Assuming that **reportTestCase.py** is our python script that accepts the parameters for updating a test case in the QAWeb database, the sample code below shows how a remote SSH call would look like:

```
ssh qa-user@qaweb 'python /home/qa-user/reportTestCase.py \
--title="VLC Player Crash on streamed video" \
--desc="Trying to investigate why that happens." \
--config="Windows 7, 64-bit" \
--state=1 \
--owner="QA Tester A" \
--project=9 \
--team=9 \
--tid=1111111;
```

Finally, another last feature that was desired was the ability to receive a daily *email notification* with all the updated graphs. That was again achievable by writing a *python script* that can talk directly to the QAWeb models and by just scheduling that script to run via a cron system job. In practice, the potentials for further automation is more than adequate, since the python language is able to handle the most demanding requests.

4 Future Work & Conclusions

In this paper we presented a basic proof of concept (i.e. QAWeb platform) on how someone can use the Django Web Framework via SSH, in order to automate the generation and presentation of QA statistics. We discussed the internals of the QAWeb platform and we identified why we consider our solution to be attractive and extendable enough. As a future work for QAWeb we would recommend a feature that will allow maintaining history for QA statistics and/or an extension that will allow QAWeb to be used for Unit Test reports directly from the equivalent frameworks (i.e. Google Tests).

References

[1] The Django Framework - *The Web framework for perfectionists with deadlines.* `https://www.djangoproject.com`

[2] Django Packages - Apps and more. `http://www.djangopackages.com/categories/apps`

[3] Djapian - High level Xapian integration for Django. `http://code.google.com/p/djapian`

[4] Matplotlib - *A python 2D plotting library.* `http://matplotlib.sourceforge.net`

[5] Xapian - *An Open Source Search Engine Library.* `http://xapian.org`

[6] QAWeb - *Online Demo, Download Source Code* `http://qaweb.loonydesk.com`