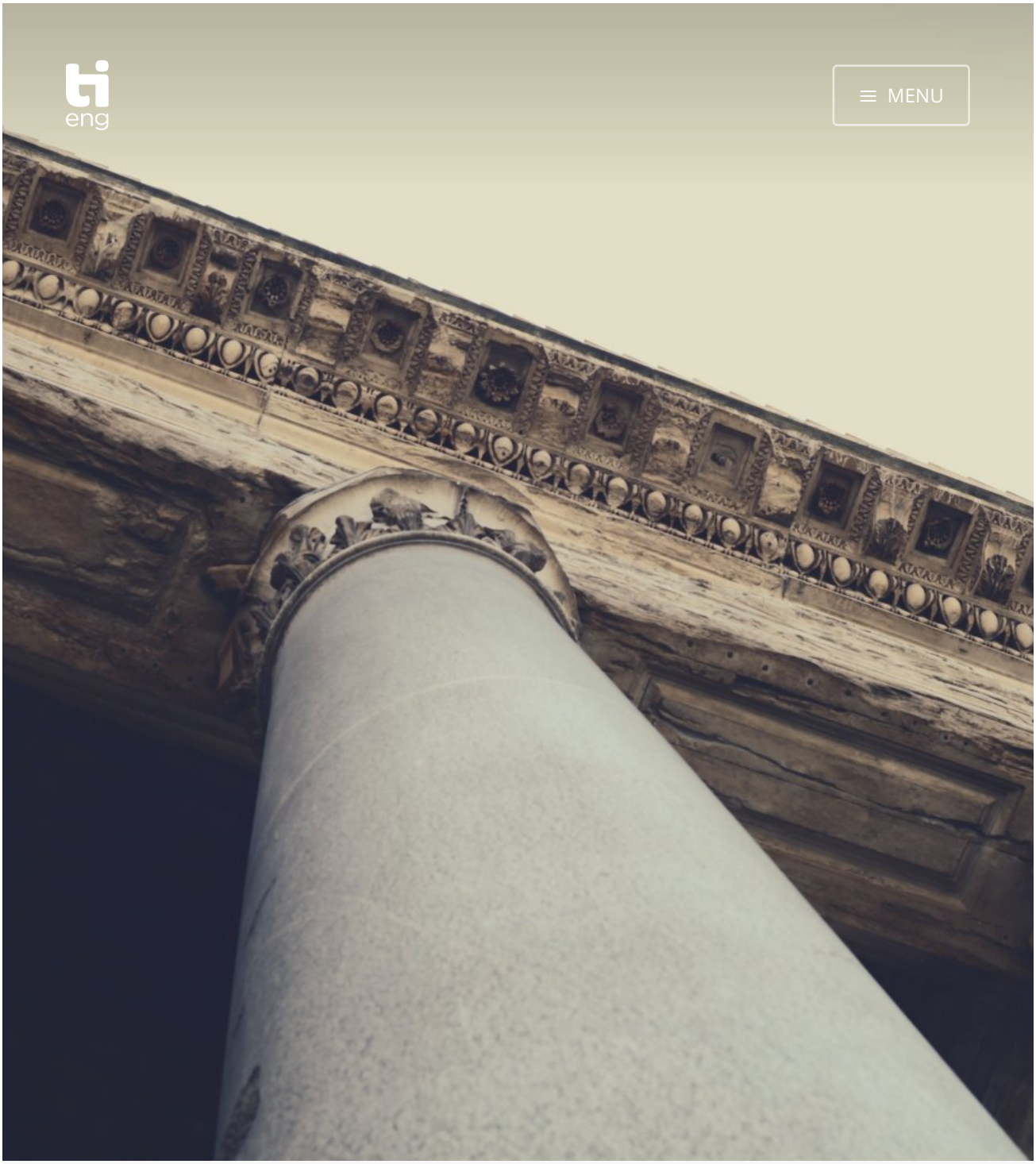


How we migrated ticketea.com to Python

3 in two weeks

12 January 2018 on python, django, cloud, gae

Ticketea.com is the platform where we showcase and sell the majority of our events. The project powering it (codenamed "Aphrodite"), was originally built using Python 2.7, and it was kept updated when new Django versions came along. It currently runs on Django 1.11.

Python 3 is the present of the Python language, and all of our new projects are being written using Python 3.6. When adding new features to ticketea.com, we often came across compatibility issues with components written for newer versions of the language. We decided not to delay the Python 2.7 → 3.6 migration anymore, and started working on it.

All of our projects come with a Dockerfile, so the first step was to change Aphrodite's base image to be `python:3.6-slim`.

This triggered the first issues:

- Outdated external libraries which needed to be updated to newer versions featuring Python 3 compatibility
- `basestring` to `str`, `urlparse` to `urllib.urlparse` and similar major changes
- Dictionary change like `iteritems()` to `items()`, or

`.items()` now returning a view.

- Things that weren't needed anymore, like Django's `force_unicode` or `__future__` library tools.

Many of these issues were corrected just by running 2to3, which not only fixed many of the compatibility issues, but also applied patterns such as rewriting `filter` + `lambda` to list comprehensions. Way safer than doing raw `grep` and `sed` replacements!

Once we finished working on the "low-hanging fruits", the next step was to run Aphrodite's test suite and achieve zero errors. We have multiple satellite Python libraries and components, and while most already had a Python 3 compatibility branch, a few were outdated and missing the latest bugfixes coming from master. Once this was fixed, we discovered a few bugs while performing integration tests. We ended up fixing things, increasing the test coverage of those components, and those Python 3 branches won't be forgotten again (and will be merged to master soon!).

Along the way, we applied the boy scout rule of "Always leaving the campground cleaner than you found it". So we fixed a few Django 1.11 deprecation warnings to prepare for Django 2.0. As it usually happens with Django, the official documentation was excellent, explaining very clearly how to implement the necessary changes.

When the test were all green, we started working on the

harder part: Manual integration tests, first in a local dev environment, then in our staging environment.

Some of the issues we faced were:

Having to keep the Python 3 branch up to date with `master`: This is a considerable effort unless you're able to freeze any change to master (very unlikely in the real world).

A non-trivial feature was delivered during the migration. Due to the amount of modified code, we thought it was easier to merge the feature branch into the Python 3 branch ASAP, and keep them synchronized.

Code coverage was around 70%, so of course when something failed it was always from that remaining 30%. There were no major issues and no need to rollback, and we have more tests as a result, but as usual, the more tests you have the better.

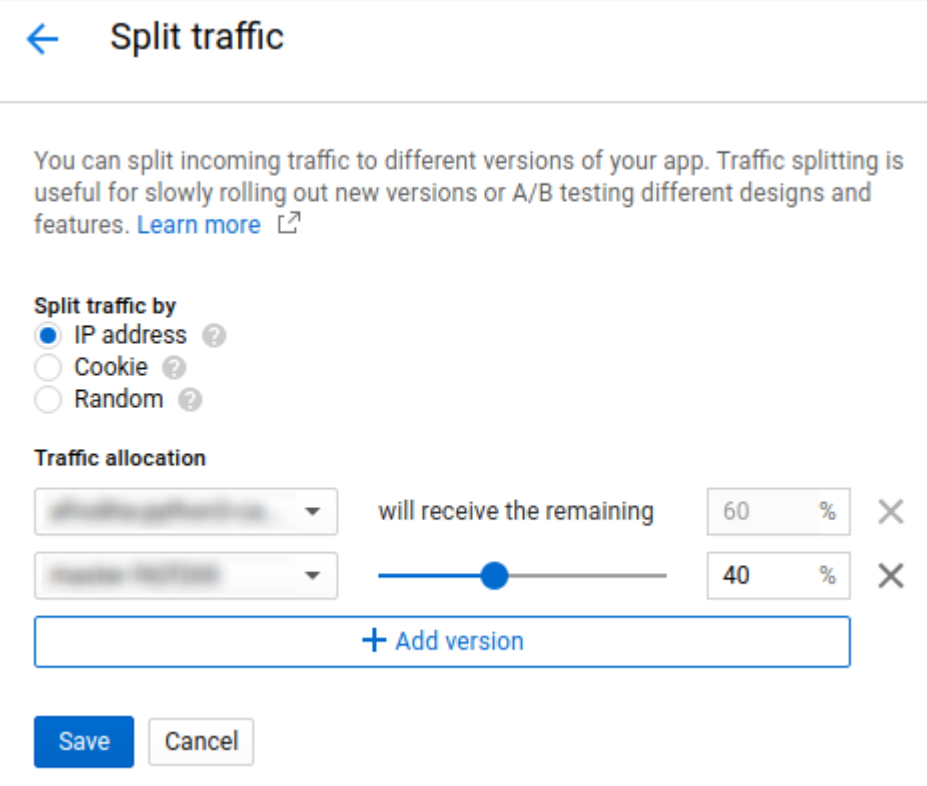
The `pickle` protocol version in python 3 can be higher than the highest available in Python 2.7. So we needed to add versioning to our Django caches, as python 2 goes only up to pickle v.2 (caches with python 3.6 serialize with pickle v.4).

Each modified file had to comply with flake8 linting rules. We recently setup a "linter test" that grabs all files you are modifying on the current branch and runs `flake8` against them. This meant that, as we had to modify around

200 files in total, all of them had to now be clean and perfectly formatted python code :) We left out Django migrations with long string literals, and fixed the rest. It took some extra effort, but the results are worth it.

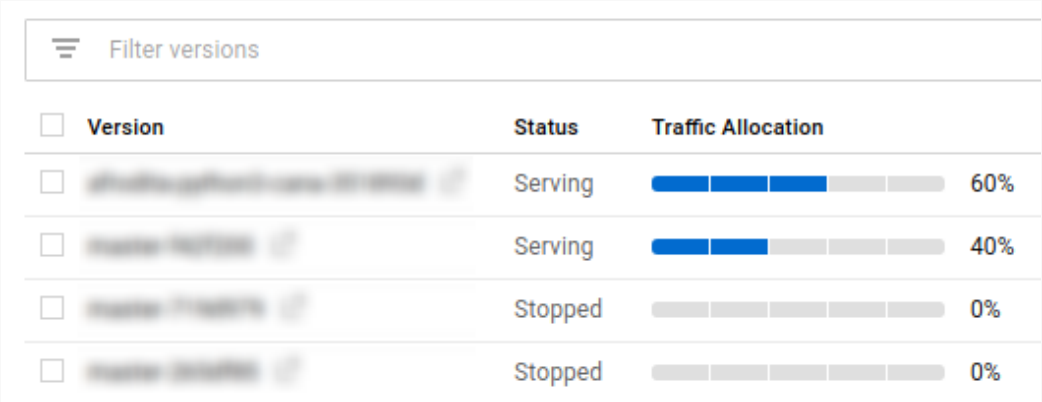
After staging tests were all fine and error logs didn't registered any issue, we proceeded to deploy to production.

Afrodita is currently running on Google App Engine Flexible, and one of the features our team loves with is **traffic splitting**:



With this feature, we can do canary releases with ease: We just deploy our new version of the service, and start redirecting small amounts of traffic traffic while we

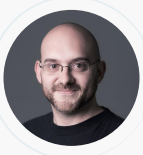
monitor for unexpected errors.



After some minor bugfixes, we could bring the traffic of the Python 3.6 version to 100% with confidence. We also had the old version available for instant rollback, thanks to how parallel versions and traffic splitting work in GAE flexible.

We finally stopped the Python 2.7 version (which was receiving zero traffic) and merged the Python 3 branch into `master`.

And that is all. As the final note, this migration was handled by just one developer and our QA engineer. When you have the right tools you can minimize overhead and focus on what really matters.

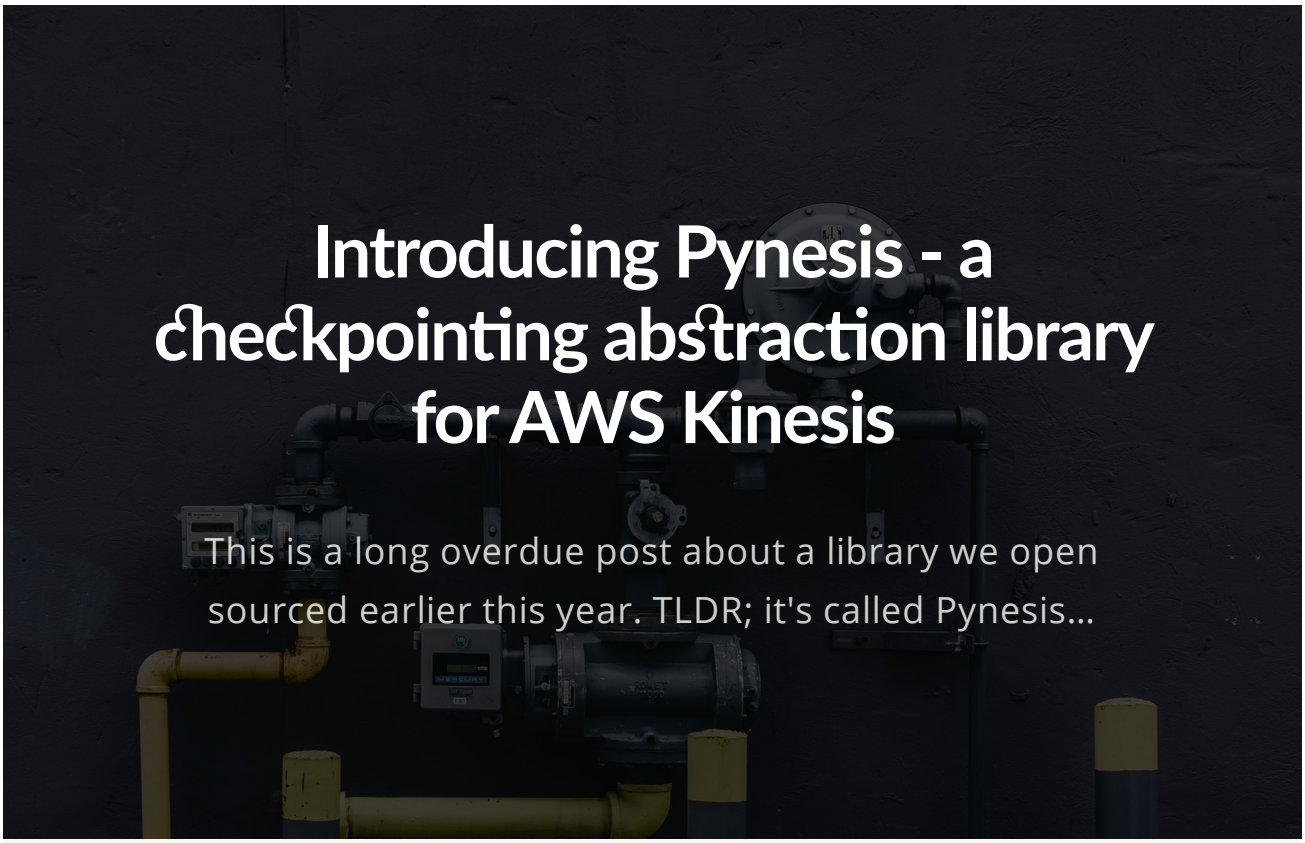


Diego Muñoz 'Kartones'

Read more posts by this author.

0.0.0.0, Madrid https://kartones.net

Share this post



Introducing Pynesis - a checkpointing abstraction library for AWS Kinesis

This is a long overdue post about a library we open sourced earlier this year. TLDR; it's called Pynesis...