There Should be One Obvious Way to Bring Python into Production

Sebastian Neubauer
sebastian.neubauer@blue-yonder.com
@sebineubauer
Agenda

• What are we talking about and why?
  • Delivery pipeline
  • Dependencies
  • Packaging
• What is the current state?
  • A walk through the different possibilities
  • Summarizing all the pros and cons
• Can we find a better solution?
  • How does the future look like?
  • Discussion: what could the „one obvious way“ be?
What are we talking about and why?
Delivery pipeline

Building/Packaging → Development → Staging/QA → Testing → Production

@sebineubauer
Delivery pipeline

Building/Packaging → Development → Staging/QA ← Testing ← Production

@sebineubauer
Development

Required:
• Fast iteration cycles, fast changes
• Automated tests can be executed

Nice to have:
• Production like local environment

Risks:
• „Works on my machine!“
• Dirty working directory
Delivery pipeline

Building/Packaging → Development → Staging/QA → Testing → Production
Building/Packaging

Required:
• Build once, use everywhere
• Possibility to compile for the target systems
• Build uniquely versioned, signed packages

Nice to have:
• Upload to an artifact repository

Risks:
• Misconfiguration of the build environment
Delivery pipeline

Building/Packaging → Development → Staging/QA → Testing → Production

@sebineubauer
Testing

Required:
• Automated
• Near production like conditions
• Reproducible conditions
• Minimal changes for testing reasons

Nice to have:
• Fast feedback
• Running after each commit on all branches

Risks:
• the tests test the test environment, but not production

@sebineubauer
Delivery pipeline

@sebineubauer
Staging/QA

Requirement:
• Automated deploy in production like environment
• Nearly no changes for testing purposes

Nice to have
• A real clone of the production system
• Possibility to run A/B tests on that system

Risks:
• outdated, manually maintained setup
Delivery pipeline

@sebineubauer

Development

Building/Packaging

Staging/QA

Testing

Production
Production

Required:
• No compiler
• No internet
• Health monitoring

Nice to have:
• Automated deploy
• Automatic monitoring
• Automatic self-healing
• Automatic rolling update and roll back

Risks:
• your business is going down…
Good setup

Continuous integration server

Building/Packaging

Modern cluster scheduler

Staging/QA

Testing

Developer’s Box

Development

Production

@sebineubauer

BlueYonder
Best decisions, delivered daily
Bad setup

Developer’s Box

Building/Packaging

Development

Staging/QA

Testing

Snowflake pet server

Production

@sebineubauer
Dependencies

“All shared software components that need to be present in the correct version so that the application works correctly”

fictitious definition
Dependency Hell

Problems:

• Transitive dependencies can have conflicting version requirements

• Python only knows application “global“ dependencies (javascript has local dependencies)

• Pip (still) doesn’t have proper dependency resolution (gh #988 open since 11 Jun 2013, but GSoC 2017 project, fingers crossed)

• System python dependencies interfere with application dependencies
Package management in python

package manager: pip
package format: wheel

- still much confusion around setuptools, distutils, eggs…
- many „best practices“ in stack overflow & co. outdated
- no standard templating for packages: see pyscaffold, versioneer…
- feels like lack of interest in the community…
- but: it has gotten way better in the last years:
  - setup.cfg
  - setuptools_scm

For details see: https://ep2017.europython.eu/conference/talks/python-packaging-current-state-and-overview by @webGandi
# Package manager hell

<table>
<thead>
<tr>
<th>System dependencies</th>
<th>Language dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>operating system, libraries</td>
<td>language specific libraries, frameworks</td>
</tr>
<tr>
<td>yum, apt-get, homebrew, vcpks…</td>
<td>pip, npm, conan, cpan, maven, composer,</td>
</tr>
<tr>
<td></td>
<td>cargo, godep, gem,…</td>
</tr>
<tr>
<td>frequent security updates</td>
<td>almost no security updates</td>
</tr>
<tr>
<td>„operations“ take care</td>
<td>„developers“ take care</td>
</tr>
<tr>
<td>root/system wide</td>
<td>user space/virtualenv</td>
</tr>
</tbody>
</table>

*Operating systems libraries take care of system-wide dependencies, while language-specific libraries are handled by developers.*

@sebineubauer
<table>
<thead>
<tr>
<th>System dependencies</th>
<th>Language dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>operating system, libraries</td>
<td>language specific libraries, frameworks</td>
</tr>
<tr>
<td>yum, apt-get, homebrew, vcpks…</td>
<td>pip, npm, conan, cpan, maven, composer, cargo, godep, gem,…</td>
</tr>
<tr>
<td>frequent security updates</td>
<td>almost no security updates</td>
</tr>
<tr>
<td>„infrastructure“ takes care</td>
<td>„developers“ take care</td>
</tr>
<tr>
<td>root/system wide</td>
<td>user space/virtualenv</td>
</tr>
</tbody>
</table>

@sebineubauer
Package manager hell

Where does it come from historically?

- disk space and bandwidth expensive
- separation between dev and ops
- single language environments
- rise of open source and sharing culture
- no package manager solved everything
What is the current state?

a walk through the different possibilities
The classical approach

Development environment:
• building proper python package (e.g. https://github.com/blue-yonder/pyscaffold)
• get everything somehow working: vagrant, conda, compile yourself…
• pushing source to git
The classical approach

On Jenkins:
• building artifacts,
• testing,
• release: packaging (wheels) and publishing to an pypi compatible artifact repository (artifactory, devpi…)
The classical approach

In production:
- standard virtualenv and pip
- application gets installed from repo together with dependencies
- OS and system dependencies are maintained separately
The classical approach

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>„standard approch“</td>
<td>dependencies are resolved in production again and again</td>
</tr>
<tr>
<td>good and supported tooling</td>
<td>need to build und upload wheels for all binary packages to</td>
</tr>
<tr>
<td>well understood</td>
<td>repository</td>
</tr>
<tr>
<td></td>
<td>because the dependencies are resolved „at runtime“,</td>
</tr>
<tr>
<td></td>
<td>developers must not forget to pin the dependencies</td>
</tr>
<tr>
<td></td>
<td>python only</td>
</tr>
</tbody>
</table>
The „conserve virtualenv“ approach

• idea: build a virtualenv, then pack it, ship it and unpack on the target system
• several similar implementations:
  • platter: simple virtualenv and wheels
  • pex: new virtualenv implementation, includes executed command
  • dh-virtualenv: virtualenvs packaged in debian packages
• done once in build step
# The „conserve virtualenv“ approach

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>• no resolving of dependencies on target host</td>
<td>• system packages not included</td>
</tr>
<tr>
<td>• no dependency to a pypi server</td>
<td>• need to compile for the exact target system</td>
</tr>
<tr>
<td>• „push or pull model“ possible, either you copy the archive to the</td>
<td>• no standard repository:</td>
</tr>
<tr>
<td>target, or it pulls from a repo</td>
<td>• implement push infrastructure</td>
</tr>
<tr>
<td>• depending on the implementation (e.g. platter) it integrates</td>
<td>• implement a repository (e.g. s3)</td>
</tr>
<tr>
<td>well in „standard“ workflow with standard tools</td>
<td>• python only</td>
</tr>
</tbody>
</table>
The OS package approach

- idea: package the application as a standard OS package, e.g. debian package
- this way you can install the application with "apt-get install" on the target machines
- deb package building is done once in the build step
- there are some few tools that help you:
  - stdeb: build deb packages with one command (can’t get it to work, last commit 2 years ago :face_with_rolling_eyes:)
  - dh-virtualenv
- for all dependencies, you either have to make deb packages too, or you bundle them up (see dh-virtualenv)
# The OS package approach

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>• integrates well with system maintenance</td>
<td>• tooling seems to be very badly maintained</td>
</tr>
<tr>
<td>• just one package manager needed</td>
<td>• no tooling for dependency management, you have to</td>
</tr>
<tr>
<td>• standard debian repository</td>
<td>declare the dependencies yourself (or use dh-virtualenv)</td>
</tr>
<tr>
<td></td>
<td>• you need a debian repository</td>
</tr>
<tr>
<td></td>
<td>• working with deb packages is often: globally</td>
</tr>
<tr>
<td></td>
<td>installed by root, not always what one needs</td>
</tr>
</tbody>
</table>
The „container as PM“ approach

Developer’s box:
• download a base image
• provision the base image
• develop the application in the container
• commit the scripts for the provisioning and deploy in the container
The „container as PM“ approach

On Jenkins:
• build the container image with the application baked in, using the scripts
• run the tests inside the container
• if all tests pass, upload the image to the registry (artifactory, docker registry…)

@sebineubauer
In production:
• let the target hosts pull the image from the registry or push it to the hosts
• start it
# The „container as PM“ approach

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>good understood technology, de-facto industry standard: schedulers, repositories, monitoring</td>
<td>security updates on host irrelevant for application</td>
</tr>
<tr>
<td>complete decoupling from host OS, windows, mac, jenkins or coreos in production, the application runs in the <code>==</code> same environment</td>
<td>without proper processes and tooling, easy to do it wrong: unapproved software in production, heartbleed…</td>
</tr>
<tr>
<td>complete environment+application is built once</td>
<td>dependency resolution and pypi server still needed</td>
</tr>
<tr>
<td>everything in git</td>
<td>chasm between system and language dependencies still exists, but now in a container</td>
</tr>
<tr>
<td>language independent</td>
<td>doesn’t really reduce complexity</td>
</tr>
</tbody>
</table>
The „next packet manager“ approach

There are many (interesting) other package managers out there:

Conda:
• Python, R, Scala, Java, Javascript, C/ C++, FORTRAN
• also packages system dependencies
• works flawlessly together with pip
• easy to use
• mature
• so far no real on premise repository, but easy to implement
The „next packet manager“ approach

Nix:
• really interesting concept: purely declarative functional language for expressing dependencies
• immutable and git like behavior: uninstallation is a well defined rollback/revert
• NixOS: completely removes the chasm between system and language dependencies
• language independent
• lazy evaluation: dependencies only get installed if needed
• not production ready yet…I guess…
The „next packet manager“ approach

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>• in the end it is a „package manager“ problem, so maybe there is a „next package manager“ that solves most of the problems</td>
<td>• package management is only part of the problem:</td>
</tr>
<tr>
<td>• there are package managers that solve parts of the problems even today (e.g. get numeric python packages working on mac and on windows using conda…)</td>
<td>• security updates</td>
</tr>
<tr>
<td>• a good end to end, language agnostic package management solution has a huge potential</td>
<td>• auditing</td>
</tr>
<tr>
<td></td>
<td>• same environment for development, testing and production</td>
</tr>
<tr>
<td></td>
<td>• no end to end solution so far</td>
</tr>
<tr>
<td></td>
<td>• very hard to get the critical mass needed that it is a holistic solution for the whole problem for all languages</td>
</tr>
</tbody>
</table>
The vending approach

• Instead of depending on external libraries, you copy the source code into your repository
• you don't have any requirements (at least in your language)
• you build just one big application package in one go
• on the target system, you install one package with no dependencies
• slightly similar to the „conserve virtualenv“ approach
## The vending approach

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>• no dependency resolution at all</td>
<td>• no dependency resolution at all, everything needs to be done manually</td>
</tr>
<tr>
<td>• easy IDE code discovery</td>
<td>• hard work to keep it up to date</td>
</tr>
<tr>
<td>• no dependency to external repositories</td>
<td>• easier to patch third party libraries instead of contributing and wait for release</td>
</tr>
<tr>
<td>• easy to patch third party libraries</td>
<td>• dangerous licensing issues</td>
</tr>
<tr>
<td></td>
<td>• useless for library development</td>
</tr>
</tbody>
</table>
Can we find a better solution?
How does the future look like?

Containers are here to stay, for many reasons
How does the future look like?

DevOps is the working mode
How does the future look like?

Polyglott: the right language for the job
How does the future look like?

Open source/sharing of code is increasing
How does the future look like?

Automation is a must

@sebineubauer
How does the future look like?

After that problem is solved, „serverless" becomes a thing
And now a short discussion!
Discussion

OS Package manager

Conserve virtualenv

Classical/wheels

Container

Next package manager

Vendoring

Other??
Q&A
Attributions

By BrokenSphere (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

By Rl (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html), CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/) or CC BY-SA 2.5-2.0-1.0 (http://creativecommons.org/licenses/by-sa/2.5-2.0-1.0)], via Wikimedia Commons