Plone Jenkins/CI Team

Release 1.0

Nov 14, 2018

Contents

CHAPTER 1

PLONE JENKINS/CI TEAM

1.1 Who's on the team

- Ramiro B. da Luz
- Jonas Baumann
- William Deegan
- Thomas Desvain
- Gil Forcada
- Tom Gross
- Ed Manlove
- Ramon Navarro Bosch
- Ross Patterson
- Martin Peeters
- Asko Soukka
- Eric Steele
- Timo Stollenwerk
- Héctor Velarde

1.2 Who leads it

Timo Stollenwerk < tisto @ plone.org>

1.3 What is the mission of the team

- Running automated software tests and code analysis for Plone core, Plone core projects and selected Plone add-ons
- Notifying developers about regressions and code quality

1.4 What is the team doing, why the team exists

- What we are doing: running jenkins.plone.org
- Why do we exist: to ensure a high software quality for Plone

1.5 How to contact the team

- We currently use the plone-website mailinglist: `plone-website@lists.sourceforge.net`

1.6 How the activities are organized

- See https://github.com/plone/jenkins.plone.org and https://github.com/plone/jenkins.plone.org/issues for current activities
- We currently do not have regular meetings

1.7 How to become part of the team

- Send an email to tisto @ plone.org or ping me on irc (tisto)

CHAPTER 2

Servers

http://jenkins.plone.org runs on a number of different servers, see below.

2.1 Master

- hosted at hetzner.de
- IP: 78.47.49.108
- donor: Timo Stollenwerk
- contact: Timo Stollenwerk (tisto) and Gil Forcada (gforcada)

2.2 Nodes

2.2.1 Nodes server 1

- hosted at hetzner.de
- IP: 88.99.26.113 / 2a01:4f8:10a:2ae::2
- donor: Plone Foundation
- contact: Paul Roeland (polyester) and Gil Forcada (gforcada)

2.2.2 Node 4

- hosted at hetzner.de
- IP: 46.4.157.69
- donor: Jens Klein

- contact: Jens Klein (jensens) and Gil Forcada (gforcada)

2.2.3 Nodes server 2

- hosted at hetzner.de
- IP: 136.243.46.143 / 2a01:4f8:212:e8c::2
- donor: Plone Foundation
- contact: Paul Roeland (polyester) and Gil Forcada (gforcada)

2.2.4 Nodes server 3

- hosted at hetzner.de
- IP: 136.243.44.103 / 2a01:4f8:212:c5a::2
- donor: Plone Foundation
- contact: Paul Roeland (polyester) and Gil Forcada (gforcada)

Configuration

Base system: Ubuntu 18.04 LTS minimal

Install lxd:

```
apt-get install lxd
```

Initial configuration:

```
lxd init
(all default options)
```

Be sure that enough space is given! By default LXD from Ubuntu 18.04 creates a loop device with only ~30Gb of space, if that's the case, do the following:

```
truncate -s100G /var/lib/lxd/disks/more-space.img
ld=$(losetup --show --find /var/lib/lxd/disks/more-space.img); echo "$ld"
lxc storage create more-space btrfs source="$ld"
```

Create nodes:

```
lxc launch ubuntu:18.04 node1 -s more-space
lxc launch ubuntu:18.04 node2 -s more-space
lxc launch ubuntu:18.04 node3 -s more-space
```

Note: The `-s` parameter with its value are not needed, if the default storage is big enough already.

Add SSH keys:

```
lxc file push /root/.ssh/authorized_keys node1/root/.ssh/authorized_keys
lxc file push /root/.ssh/authorized_keys node2/root/.ssh/authorized_keys
lxc file push /root/.ssh/authorized_keys node3/root/.ssh/authorized_keys
```

Write down nodes IPs:

```
lxc list
```

Configure a jump host to connect to them:

```
Host jenkins-plone-org-nodes-host
  HostName 88.99.26.113
  User root
  ProxyCommand none

Host node1-jenkins-plone-org
  HostName XX.XX.XX.XX
  User root
  ProxyCommand ssh jenkins-plone-org-nodes-host nc %h %p 2> /dev/null

Host node2-jenkins-plone-org
  HostName XX.XX.XX.XX
  User root
  ProxyCommand ssh jenkins-plone-org-nodes-host nc %h %p 2> /dev/null

Host node3-jenkins-plone-org
  HostName XX.XX.XX.XX
  User root
  ProxyCommand ssh jenkins-plone-org-nodes-host nc %h %p 2> /dev/null
```

Connect to all nodes to accept their fingerprint:

```
ssh node1-jenkins-plone-org
ssh node2-jenkins-plone-org
ssh node3-jenkins-plone-org
```

Add iptables rules to let jenkins master connect to the nodes, these two lines are needed **for each** node:

```
iptables -t nat -A PREROUTING -p tcp --dport ${SPECIFIC_PORT} -j DNAT --to-
↪destination ${NODE_IP}:22
iptables -t nat -A POSTROUTING -p tcp -d ${NODE_IP} --dport ${SPECIFIC_PORT} -j SNAT -
↪-to-source ${SERVER_IP}
```

Note: update SPECIFIC_PORT to something like 808X (each node a different port), NODE_IP to the IP of each node (node IP can be seen with `lxc list`) and SERVER_IP to the server host (i.e. 88.99.26.113)

TODO

- create ansible playbook for bootstrap the server so it does:
  - create containers with ansible
  - configure SSH
  - configure firewall

CHAPTER 3

jenkins.plone.org Set Up Howto

This document describes how to set up the entire Jenkins infrastructure for jenkins.plone.org. Those are the main steps:

- Set up Jenkins server (jenkins.plone.org, with Ansible)
- Set up Jenkins nodes (node[1-x].jenkins.plone.org, with Ansible)
- Set up the Jenkins jobs on the Jenkins server (with Jenkins Job Builder)

3.1 Prerequisites

Checkout this repository:

```
git clone git@github.com:plone/jenkins.plone.org.git
cd jenkins.plone.org
```

Create and activate a virtualenv:

```
python3.6 -m venv .
./bin/activate
```

Install all the tools needed (ansible, ansible roles and jenkins-job-builder):

```
pip install -r requirements.txt
ansible-galaxy install -r ansible/roles.yml
git submodule update --init
```

Note: For the roles that are downloaded from checkouts, plone.jenkins_server and plone.jenkins_node, you will need to remove them and clone them manually if you want to make changes on them.

```
cd ansible/roles
rm -rf plone.jenkins_server
rm -rf plone.jenkins_node
git clone git@github.com:plone/plone.jenkins_server
git clone git@github.com:plone/plone.jenkins_node
```

Check ansible/inventory.txt and make sure that you can connect to the machines listed there.

Copy your public ssh key to all servers:

```
ssh-copy-id -i ~/.ssh/<SSH-KEY>.pub root@<SERVER_IP>
```

3.1.1 Set Up Jenkins Server

```
./update_master.sh
```

3.1.2 Set Up Jenkins Nodes

```
./update_nodes.sh
```

3.1.3 Set Up Jenkins Jobs

Do the steps described above to clone, activate virtualenv and fetch submodules.

Put jenkins-job-builder in development mode:

```
cd src/jenkins-job-builder
pip install -r requirements.txt -c ../../requirements.txt
python setup.py develop
```

Test the jobs are properly setup:

```
jenkins-jobs --conf jobs/config.ini.in test jobs/jobs.yml -o output
```

Note: A folder named `output` should contain one file per each jenkins job configured on jobs.yml

Create your own `jobs/config.ini` by copying it from `jobs/config.ini.in`:

```
cp jobs/config.ini.in jobs/config.ini
```

Add your own credentials to jobs/config.ini. You can find them when you log into Jenkins and copy your API token (e.g. http://jenkins.plone.org/user/tisto/configure).

Create your own `ansible/secrets.yml` by copying it from `ansible/secrets.yml.in`:

```
$ cp ansible/secrets.yml.in ansible/secrets.yml
```

Add github API secrets that are needed for the github login functionality on jenkins.plone.org. You can find those settings on plone organization in github: https://github.com/organizations/plone/settings/applications

Look for the `Plone Jenkins CI` application name.

For the `github_api_key` you need a personal token (from https://github.com/jenkins-plone-org github user).

Now finally install the jobs on the server:

```
./update_jobs.sh
```

3.1.4 Manual Configuration

There are currently a few steps that we need to carry out manually. We will automate them later.

1. Github post-commit hook for buildout.coredev:

- go to https://github.com/plone/buildout.coredev/settings/hooks
- create a new webhook with the following details:
  - Payload URL: http://jenkins.plone.org/github-webhook/
  - Content type: application/x-www-form-urlencoded
  - Secret: *nothing*
  - Which events would you like to trigger this webhook?: Send me everything
  - Active: yes

2. Manage Jenkins -> Configure System:

- E-mail Notification:
  - SMTP Server: smtp.gmail.com
  - Use SSL: True
  - SMTP Port: 465
  - Reply-To Address: jenkins@plone.org
  - Use SMTP Authentication: True
    - User Name: jenkins@plone.org
    - Password: ...

3. Manage Jenkins -> Manage Credentials -> Add Credentials: SSH Username with private key:

- Scope: System
- Username: jenkins
- Description: jenkins.plone.org private ssh key
- Private Key: From a file on Jenkins master: File: /var/lib/jenkins/jenkins.plone.org

=> Upload jenkins.plone.org private ssh key manually to /var/lib/jenkins => chown jenkins:jenkins jenkins.plone.org

CHAPTER 4

Testing locally (vagrant)

As collaborative configuration of servers is really useful to spread knowledge, so it is also that one can step on each others toes, or not be sure if a server can be really be used for testing, or if others are *already* testing on it while you also want to test...

So to make it short and easy: having a local environment (i.e. virtual image) to test changes before sending a pull request is extremely helpful.

For this vagrant is a perfect fit.

4.1 Set up

- install vagrant and its dependencies and everything said on *jenkins.plone.org Set Up Howto*
- from within a clone of jenkins.plone.org repository checkout run:

```
vagrant up master
vagrant up node
```

`vagrant` will automatically run the ansible playbook for them.

If the playbook fails, run:

```
vagrant provision master
vagrant provision node
```

And that will re-run the ansible playbook once again, and hopefully fixing the previous problem.

You can enjoy your newly jenkins.plone.org master server locally at:

`http://localhost:8080`

Note: There seems to be some problems with nginx.

POST requests get the port (8080) removed on the response. Adding the port back on the wrong URL makes it work again.

Finally, to run `jenkins-job-builder` on it, run:

```
jenkins-jobs --conf jobs/config.ini.in update jobs/jobs.yml
```

Enjoy!

CHAPTER 5

Run pull request jobs

Before merging a pull request on GitHub (or manually via the command line), one needs to be sure that nothing breaks due to the changes made on the pull request.

For that we have some special jenkins job meant for that:

- If the pull request targets Plone 5.2: http://jenkins.plone.org/job/pull-request-5.2
- If the pull request targets Plone 5.1: http://jenkins.plone.org/job/pull-request-5.1
- If the pull request targets Plone 5.0: http://jenkins.plone.org/job/pull-request-5.0
- If the pull request targets Plone 4.3: http://jenkins.plone.org/job/pull-request-4.3

If the pull request targets **both** 5.2 *and* 5.1 you need to run a job on each of the jenkins jobs mentioned above.

5.1 Test a pull request

To test a pull request with this jenkins job, either:

- watch this video: https://youtu.be/mXs_OcJhjnU
- follow the detailed steps below

5.1.1 Run a pull request job

- go to http://jenkins.plone.org
- log in with your github user
- click on the Pull Request 5.2 job or Pull Request 5.1 job or Pull Request 5.0 job or Pull Request 4.3 job if you are targeting that Plone version
- click on the huge button **Build with Parameters** Plone 5.2 or Plone 5.1 or Plone 5.0 or Plone 4.3
- paste the pull request URL on the text field (or multiple pull requests if they have to be combined, then one per line), like for example https://github.com/plone/plone.outputfilters/pull/16

- click on the `Build` button

5.2 GitHub integration

As soon as the job starts the pull request on GitHub will be notified, showing that the pull request is being tested by jenkins.plone.org.

When it finishes, GitHub will be notified again and either report that all tests passed, or that there has been some failures and thus it would not be wise to merge the pull request.

5.3 Mail integration

Reporting is not only done via GitHub, but also by email.

When the jenkins job is finished it will report by mail to the user that started the jenkins job.

On the mail, the status of the job and a link to the jenkins job itself and to the pull request(s) are provided for convenience.

CHAPTER 6

Run code analysis on a package

jenkins.plone.org is running code analysis (via plone.recipe.codeanalysis) on a number of packages, see the up-to-date list.

Follow these steps to check how any arbitrary package adheres to our Plone official guidelines.

Note: See the very same script that Jenkins uses, below follows a more detailed step to step on how to run it and fix the errors.

Clone the repository and create a Python virtual environment:

```
git clone git@github.com:plone/plone.app.discussion.git
cd plone.app.discussion
virtualenv .
source bin/activate
```

Create a cleanup branch, although not mandatory it's always a good idea:

```
git checkout -b cleanup
```

Get the QA configuration and bootstrap:

```
wget https://raw.githubusercontent.com/plone/buildout.coredev/5.2/bootstrap.py -O␣
↪bootstrap.py
wget https://raw.githubusercontent.com/plone/buildout.coredev/5.2/experimental/qa.cfg␣
↪-O qa.cfg
wget https://raw.githubusercontent.com/plone/plone.recipe.codeanalysis/master/.isort.
↪cfg -O .isort.cfg

python bootstrap.py --setuptools-version 31.1.1 --buildout-version 2.8.0 -c qa.cfg
```

Adjust `qa.cfg` to the package:

- check that the `directory` option on `code-analysis` part matches the top-level folder of the distribution

- remove the `jenkins = True` line (so that `bin/code-analysis` shows its report on the terminal)

Finally run buildout and code analysis:

```
bin/buildout -c qa.cfg
bin/code-analysis
```

The first easy fixes can be easily solved with `autopep8` and `isort`:

```
pip install autopep8 isort

isort plone/app/discussion/*.py
autopep8 --in-place -r plone/app/discussion
```

By default autopep8 does white space only changes which are basically guaranteed safe.

Important exception: undo any changes made by autopep8 to Python **skin** scripts. For instance, it will change the double comment hashes at the top to single hashes, which completely break those Python scripts.

After committing the initial autopep8 run, you can run autopep8 in more aggressive mode, but you have to check these changes more carefully:

```
autopep8 --in-place --ignore W690,E711,E721 --aggressive
```

Keep running `bin/code-analysis` to see how much errors are still left to be fixed.

Once finished, add a comment on `CHANGES.rst` and commit all the changes in a single commit:

```
$EDITOR CHANGES.rst

git commit -am"Cleanup"
```

Push the branch:

```
git push -u
```

Create a pull request on github and start a jenkins job to verify that your changes did not break anything. For that, see the *docs about testing pull requests*.

Lastly file an issue on jenkins.plone.org issue tracker so that Jenkins start monitoring the package.

Done! Thanks for cleaning one package!

CHAPTER 7

Run add-on jobs

Before a final release of Plone core is done, add-ons might want to check if they need any porting effort to make the add-on compatible with it.

For that we have some special jenkins jobs:

- If the add-on targets Plone 5.1: http://jenkins.plone.org/job/test-addon-5.1
- If the add-on targets Plone 5.0: http://jenkins.plone.org/job/test-addon-5.0
- If the add-on targets Plone 4.3: http://jenkins.plone.org/job/test-addon-4.3

7.1 Test an add-on

- go to http://jenkins.plone.org
- log in with your github user
- click on the Test add-on against Plone 5.2 job or Test add-on against Plone 5.1 job or Test add-on against Plone 5.0 job or Test add-on against Plone 4.3 job if you are targeting that Plone version
- click on the huge button **Build with Parameters** Plone 5.2 or Plone 5.1 or Plone 5.0 or Plone 4.3
- paste the add-on git URL for the add-on that you want to test on the `ADDON_URL` field, for example https://github.com/collective/collective.cover.git
- *(optionally)* type the branch you want to test the add-on against on the `ADDON_BRANCH` field, by default it will be the master branch
- click on the `Build` button

Note: For the jobs to work properly they need to get the add-on name out of the URL, for that, the last path of the URL will be used, i.e. https://github.com/my-org/my-cool-repo.git or https://gitlab.com/another-org/project/something/else/my-cool-repo.git If not the add-on name can not be guessed with a regular expression.

7.2 GitHub integration

A comment will be added on the latest commit on that branch once the job finishes.

7.3 Mail integration

When the jenkins job is finished it will report by mail to the user that started the jenkins job.

On the mail, the status of the job will be provided.

CHAPTER 8

Check package dependencies

Systems evolve over time, new ideas come in and old ones go away.

As simple and logic as it reads, the problem with the sentence above is the *going away* part.

To remove/deprecate a complete package (say `Products.ATContentTypes`), one first needs to know where it is being used, then, one by one, keep updating all packages to remove that dependency.

Only after *all packages* have removed all traces of that package, it can be safely removed.

But wait, during that time, could it be that someone reintroduced a reference to that package?

Or to begin with, how do we know which packages are the ones that need to be ported?

I'm glad you ask.

By carefully keeping the list of packages needed by a package, and a way to keep track if new dependencies are added or removed, one can then really be sure that a package is free from another one.

8.1 Tracking dependencies

In Python there is usually always a tool available, also for dependencies tracking: enter z3c.dependencychecker.

`z3c.dependencychecker` is a tool that scans a package structure (after its `egg_info` directory is built, either by buildout or directly calling `setup.py`) and generates various reports regarding dependencies.

It not only tells you which dependencies are missing in `setup.py`, but also which dependencies are not needed, which dependencies are missing for tests, and which ones should be only test dependencies rather than general dependencies.

As life is complex, there are quite a few packages that can not be correctly identified, or packages that are soft dependencies, and thus they should not be made mandatory.

For these cases, `z3c.dependencychecker` also has an answer: `.pyproject.toml`.

In this configuration file, expected to be found on the repository top level, a table (in TOML parlance) is expected to help `z3c.dependencychecker` know about these corner cases.

For example, given this content:

```
[tool.dependencychecker]
ignore-packages = ['plone.app.dexterity' ]
Zope2 = ['Products.Five', 'Products.OFSP']
ZODB = ['ZEO', 'Pizza']
```

Any reference to `plone.app.dexterity` will not be reported, while any references to `Products.Five` and `Products.OFSP` will be treated as if they were `Zope2` references, and the same goes for `ZODB`, `ZEO` and `Pizza`.

With all these, one can get ready to clean a package dependencies.

See z3c.dependency on pypi for more details.

8.2 Clean a package

On Plone Jenkins there is a Jenkins Job that reports if a package has its dependencies correctly defined.

You can use it to check what's the status of a package, or to verify that the changes you made on it are indeed correct.

To use it, follow this instructions:

Log in with your GitHub account in Jenkins and click on the `Build with Parameters` button on the job page.

A form with two fields will show up: type the package name (i.e. `plone.app.dexterity`) and the branch to test (i.e. `master`) and click on the `Build` button.

A Jenkins job will be started.

As soon as it finishes, it will report by email the `z3c.dependencychecker` report to you.

Take that report and fix the package's `setup.py`, add the `.pyproject.toml` on it with the `[tool.dependencychecker]` (even if it is no needed) and make a pull request with everything.

As soon as there is this `.pyproject.toml` with the `[tool.dependencychecker]`, Jenkins (via mr.roboto) will start automatically that job and report back on the pull request itself.

8.2.1 Work locally

To check the random status of a package, or simply if a package needs clean up, the jenkins job is indeed useful.

But to actually work on cleaning a package it can be a bit cumbersome, as you have to keep switching from your editor+terminal, the browser to check Jenkins output and your email client to get the reports.

For that a new `part` in `buildout.coredev` GitHub project was added: `dependencies`.

To clean a package do the following:

- get `buildout.coredev` repository
- switch to branch 5.2
- remove all checkouts and leave/add only the package that you want to clean up on `checkouts.cfg`
- virtualenv + bootstrap + buildout
- run dependencychecker on the package
- repeat the last two steps (see below) until the output is completely clean up

```
git clone https://github.com/plone/buildout.coredev
cd buildout.coredev
git checkout 5.2
${EDITOR} checkouts.cfg
virtualenv .
pip install -r requirements.txt
buildout -c core.cfg install dependencies
./bin/dependencychecker src/${PACKAGE}
```

Note: The idea of changing `checkouts.cfg` is to speed up the buildout run.

For that, the `install` command on buildout allows also to only install a single part, thus, further speeding up the process.

Congratulations, you helped cleaning up a package!!

CHAPTER 9

Indices and tables

- genindex
- modindex
- search