НИИТ

Лекция №15

Развертывание и управление

- setuptools
- Управление развертыванием ПО
- Системы управления конфигурацией
- Puppet
- Chef
- Ansible
- SaltStack
- Практика

python

setuptools

Программное обеспечение должно иметь удобный формат для развертывания на компьютерах конечных пользователей. Python предлагает для этих целей библиотеку setuptools, позволяющую компоновать все файлы программы в пакеты, с которыми может работать установщик pip, и даже регистрировать эти пакеты в глобальном хранилище PyPI.

Как правило, проект (например, my_project) имеет следующую структуру:

```
my_project_dir/
    my_project/
        main_file.py
        any_other_file.py
    tests/
        test_main_file.py
        test_any_other_file.py
```

setuptools

Чтоб этот проект можно было упаковать в пакет, а также установить в site-packages надо дополнить проект файлами __init__.py и setup.py:

```
my_project_dir/
    my_project/
        __init__.py
        main_file.py
        any_other_file.py
    tests/
        test_main_file.py
        test_any_other_file.py
    setup.py
```

Файл __init__.py – пустой. Его наличие в папке означает, что эта папка со всем содержимым является частью проекта и должна попасть в пакет.
Файл setup.py определяет верхний уровень проекта и содержит информацию по пакету для утилиты setuptools.
Тесты (папка tests), как правило, не должны попадать в распространяемый пакет, т.к. не являются частью программного продукта.

setuptools

Содержимое setup.py может, к примеру, выглядеть так:

```
from setuptools import setup

setup(name='mycoolproject',
      version='0.0.1',
      description='Project is really cool',
      author='John Doe',
      author_email='jdoe@gmail.com',
      packages=['my_project']
      )
```

Файл обязательно должен содержать импорт setup из setuptools и поля name (имя пакета) и packages (каталоги, которые войдут в итоговый пакет). Сборка пакета осуществляется командой:

```
$ python3 setup.py sdist
```

После этого архив с пакетом mycoolproject-0.0.1.tar.gz можно найти в папке dist.

setuptools

Если для пакета необходимо устанавливать дополнительные сторонние библиотеки (иными словами пакет имеет зависимости), эти библиотеки необходимо прописать в отдельном файле requirements.txt (можно создать его рядом с setup.py). Зависимость от встроенных библиотек Python нигде указывать не нужно. Содержимое requirements.txt может быть, к примеру, таким:

```
SQLAlchemy==1.0.11
PySocks==1.6.8
click==6.7
```

Установка программы из папки с файлом setup.py может выполняться при помощи pip (опция –e заставляет pip искать устанавливаемую программу не в PyPI, а в указанной следом папке, “.” – текущая папка):

```
$ pip3 install -e .
```

Если надо использовать файл requirements.txt, он передается в качестве параметра pip с опцией –r:

```
$ pip3 install -e . -r requirements.txt
```

Управление развертыванием ПО

Быстрое развитие виртуализации вкупе с увеличением мощности серверов, соответствующих промышленным стандартам, а также доступность «облачных» вычислений привели к значительному росту числа нуждающихся в управлении серверов, как внутри, так и вне организации. И если когда-то такая система представляла собой набор стоек с физическими серверам в центре обработки данных на одном или нескольких этажах одного здания, то теперь зачастую приходится управлять гораздо большим количеством серверов (как физических. так и виртуальных), которые могут быть распределены по всему земному шару. Здесь на помощь приходят средства управления конфигурацией.

Описание проблемы

Как можно было бы попытаться решить проблему установки ПО на множество управляемых серверов. Как вариант, написать пару скриптов, в которых будет что-то наподобие:

```
# servers.sh
#!/bin/bash
servers="server00 server01 server02 server03 server04"

for server in $servers ; do
    scp /path/to/job/file/job.sh
    $server:/tmp/job.sh ssh
    $server sh /tmp/job.sh
done


# job.sh
#!/bin/bash
apt-get update
apt-get install nginx
service nginx start
```

Описание проблемы

Вроде все кажется легким и простым. Нужно что-то сделать — пишем новый скрипт, запускаем. Изменения приходят на все серверы последовательно. Если скрипт хорошо отлажен — все будет хорошо работать. До поры.

Теперь представьте, что серверов стало больше. Например, сотня. А изменение долгое — например, сборка чего-нибудь большого и страшного (например, ядра) из исходников. Скрипт будет выполняться сто лет, но это полбеды.

Что если обновление требуется выполнить только на определенной группе из этой сотни серверов. А через два дня нужно сделать другую большую задачу на другом срезе серверов. В этом случае придется каждый раз переписывать скрипты и много раз проверять, нет ли в них каких-нибудь ошибок, не вызовет ли это какой-нибудь проблемы при запуске.

Проблема в том, что в подобных скриптах описываются действия, которые необходимо выполнить для приведения системы в определенное состояние, а не само это состояние.

Системы управления конфигурацией

Системы управления конфигурацией (Configuration Management Systems) — программы и программные комплексы, позволяющие централизованно управлять конфигурацией множества разнообразных разрозненных операционных систем и прикладного программного обеспечения, работающего в них.

Современные системы управления конфигурацией по сути стремятся к тому, чтобы в полной мере реализовать принцип Infrastructure-as-Code, в соответствии с которым вся существующая IT-инфраструктура, машины, их конфигурация, связи между ними и так далее могут быть описаны одним или несколькими формальными файлами, таким образом, чтоб системы управления конфигурацией могли воплотить описанную конфигурацию в жизнь.

Системы управления конфигурацией

Очень важно тут то, что состояние всей инфраструктуры остается обозримым и контролируемым. Ручное выполнение операций на узлах минимизировано или сведено к нулю.

Это единственный путь управлять огромными и растущими инфраструктурами.

Примерами систем управления конфигурацией являются Puppet, Chef, Ansible и Salt. Они были задуманы чтобы упростить настройку и обслуживание десятков, сотен и даже тысяч серверов. Ansible и Salt написаны на Python.

Puppet

Puppet (написан на Ruby) считается наиболее используемым из четырех. Он наиболее полон с точки зрения возможных действий, модулей и пользовательских интерфейсов, представляя полную картину ЦОД, охватывая почти каждую операционную систему и предоставляя утилиты для всех основных ОС. Начальная установка относительно проста, требует развертывания головного сервера и клиентских агентов на каждой управляемой системе.

Архитектура — клиент-серверная, на сервере хранятся конфигурационные фалы (в терминах puppet они называются манифесты), клиенты обращаются к серверу, получают их и применяют.

Puppet

Манифесты Puppet декларативно описывают необходимое состояние системы, а вычисление, как к нему прийти из текущего состояния — задача самой системы управления конфигурацией. Для сравнения, манифест puppet, выполняющий ту же работу, что и пара ранее приведенных скриптов:

```
class nginx {
  package { 'nginx':
    ensure => latest
  }
  service { 'nginx':
    ensure => running,
    enable => true,
    require => Package['nginx']
  }
}
node /^server(\d+)$/ {
  include nginx
}
```

Puppet

Еще пример манифеста, проверяющего существования файла и настраивающего права доступа к нему:

```
file { '/tmp/helloworld':
    ensure => present,            # файл должен существовать
    content => 'Hello, world!', # содержимое файла - я строка "Hello, world!"
    mode => 0644,                 # права на файл - 0644
    owner => 'root',              # владелец файла - root
    group => 'root'               # группа файла - root
}
```

Chef

Chef (серверная часть написана на Erlang, клиентская – на Ruby) и остальные систему управления конфигурацией работают по схожему принципу. Основные элементы инфраструктуры Chef:

- Nodes (Ноды) — это любой сервер, физический, либо виртуальный, который вы будете настраивать с помощью Chef.
- The Server (Chef-сервер) — непосредственно сам Chef-сервер к которому обращаются клиенты (Nodes), сервер состоит из нескольких компонентов:
  - Workstations (рабочие станции) — рабочее место администратора Chef, т.е. компьютер на котором мы готовим рецепты, поваренные книги и управляем всей кухней с помощью knife.
  - Knife —это основной инструмент для работы с Chef из консоли. Именно с помощью “ножа“ мы управляем нодами и Chef-сервером.

Chef

Наиболее популярный инструмент в Chef это cookbook. В нем содержатся определенные рецепты.
Рецепт — это шаблон выполнения определённых действий на сервере с шаблонами файлов, переменными и т.п.
Можно создать свой рецепт:

```
knife cookbook create test
```

Можно добавить в рецепт “по умолчанию” директиву для установки списка пакетов, которые мы будем распространять на все наши серверы:

```
$ gedit ~/chef-repo/cookbooks/test/recipes/default.rb

%w{ntp mc htop vim-common wget curl git postfix}.each do |packages|
    package packages do
    action :install
    end
end
```

Ansible

Главное его отличие от других подобных систем в том, что Ansible использует существующую инфраструктуру SSH, в то время как другие (chef, puppet, и пр.) требуют установки специального PKI-окружения (PKI - Public Key Infrastructure – инфраструктура открытых ключей).

Преимущества ansbile:

- относительно просто освоить;
- декларативный язык описания конфигурации;
- на управляемые узлы не нужно устанавливать никакого дополнительного ПО;
- легко расширять с использованием дополнительных модулей.

Начать пользоваться ansible можно за пару минут:

```
$ sudo apt-get install ansible
$ ansible --version
ansible 2.0.0.2
  config file = /etc/ansible/ansible.cfg
  configured module search path = Default w/o override
```

Ansible

Создадим файл inventory для перечисления серверов, которыми мы хотим управлять (для начала укажем там localhost):

```
[test]
localhost ansible_connection=local
```

Теперь попробуем обратится к нашим хостам, используя ansible:

```
$ ansible -i inventory -m ping all
$ ansible -i inventory -a "ls -lah" all
```

Помимо утилиты ansible, есть еще утилита ansible-playbook, которой чаще и пользуются. Чтобы не вводить каждый раз в командную строку параметры, можно в директории проекта создать файл ansible.cfg.

```
# ansible.cfg
[defaults]
inventory = ./inventory
```

Ansible

Теперь создадим наш первый сценарий (playbook в терминологии ansible) web.yml.

```
# web.yml
---
-  hosts: all
   user: ubuntu

   tasks:
     - name: Update apt cache
       apt: update_cache=yes
       become: true

     - name: Install required packages
       apt: name={{ item }}
       become: true
       with_items:
         - nginx
         - postgresql
```

Запускаем наш первый сценарий, который обновляет кэш apt, а потом ставит два пакета: nginx и postgresql.

```
$ ansible-playbook web.yml --extra-vars "ansible_become_pass=linuxuser"
```

SaltStack

Основные элементы инфраструктуры SaltStack:

- Salt master — процесс, работающий на машинах, с которых происходит управление
- Salt minion — процесс, работающий на управляемых машинах. Процесс отвечает за коммуникацию с мастером, получение инструкций от него, их исполнение и контроль состояния своего хоста (если по каким-то причинам на хосте нельзя установить минион, команды могут исполняться просто через SSH);
- Salt state (SLS) — конфигурационные файлы, написаны на YAML (манифесты в Puppet; playbooks в Ansible; повареные книги, cookbooks, в Chef);

SaltStack устанавливается в Ubuntu при помощи apt-get:

```
$ sudo apt-get install salt-master salt-minion
```

SaltStack

Чтобы подключить minion к мастеру, нужно на миньоне указать ip мастера в /etc/salt/minion и перезапустить salt на миньоне и на мастере:

```
$ sudo /etc/init.d/salt-minion restart
[ ok ] Restarting salt-minion (via systemctl): salt-minion.service.
$ sudo /etc/init.d/salt-master restart
[ ok ] Restarting salt-master (via systemctl): salt-master.service.
$ sudo salt-key -A
The following keys are going to be accepted:
Unaccepted Keys:
VBU
Proceed? [n/Y] y
Key for minion VBU accepted.
```

Простые команды выглядят так:

```
$ salt '*' test.ping
$ salt '*' disk.usage
$ salt '*' cmd.run 'ls -l /etc'
```

SaltStack

Создаем главный state. Это файл /srv/salt/top.sls:

```
base:
  '*':
    - test_machine
```

Создаем /srv/salt/test_machine.sls:

```
/etc/testfile:
  file.managed:
    - source: salt://testfile
    - mode: 644
    - user: root
    - group: root
```

И файл testfile в /srv/salt/testfile с любым содержанием. Запускаем:

```
$ sudo salt '*' state.apply
```

SaltStack

Результат:

```
$ sudo salt '*' state.apply
VBU:
----------
          ID: /etc/testfile
    Function: file.managed
      Result: True
     Comment: File /etc/testfile updated
     Started: 22:35:44.529876
    Duration: 111.181 ms
     Changes:
              ----------
              diff:
                  New file
              mode:
                  0644

Summary for VBU
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 111.181 ms
```

Практика

* У нас есть свой сайт. Требуется написать несколько state'ов, выполнение которых приведет к установке всего сайта и его запуску с помощью SaltStack. То есть необходимо написать сценарии для установки Python нужной версии, установки нужных библиотек (можно без виртуального окружения), переносу кода сайта, запуску команды runserver и т. д. Обязательно надо проверить, все ли будет работать правильно. Последовательность инструкций устанавливается с помощью опций require и подобных.