# THE NEW NETWORK ENGINEER

## NETWORK AUTOMATION WORKSHOP

Khelil SATOR
JUNIPER NETWORKS
May 2016
Version 3.8

# AGENDA

- INTRODUCTION TO NETWORK AUTOMATION
- INTRODUCTION TO PYTHON PROGRAMMING
- BUILDING DOCUMENTS WITH JINJA2 TEMPLATES
- YAML (NON-PROGRAMMERS TO PROGRAMS)
- JUNOS AUTOMATION WITH PYEZ PYTHON LIBRARY
- JSON  DATA FORMAT (EXCHANGE DATA BETWEEN APPLICATIONS)
- PROGRAMMATIC ACCESS WITH REST APIs
- HUMAN READABLE AUTOMATION WITH ANSIBLE
- VERSION CONTROL (GIT WITH GITHUB)
- CONTINIOUS INTEGRATION (TRAVIS CI AND COVERALLS)
- VAGRANT FOR NETWORK ENGINNERS
- ONLINE NETWORK AUTOMATION QUIZ

# INTRODUCTION TO NETWORK AUTOMATION

# SOME USE CASES

- Handle configuration changes faster
  - If it is about a repetitive task. Even if it is about easy tasks.
  - Add a vlan on all your switches
  - Configure a list of interfaces with a vlan
  - Modify SNMP community over 100 devices

# SOME USE CASES

- Reduces human errors
  - Configurations are almost always deployed manually.
  - Automated configuration deployment reduces human errors and brings consistency. You can also achieve complex configuration changes with network automation (maintain a vxlan fabric, maintain an ip fabric, …)

# SOME USE CASES

- Simplify network audit
  - One way to handle this task is to manually open an SSH connection to every device and manually populate a spreadsheet with relevant information.
  - But we can perform this task in a more automated/programmatic fashion. Some examples:
    - Check if your switches uplinks ports are up
    - Check the status of your BGP neighbors (state needs to be “established”)
  - Data Collection before and after a change (a configuration change, an upgrade …)
    - Check if there is any issue after a change (do you have a BGP neighbor not in an “established” state ….)
    - Upgrade all your devices, and then check if each of them is running the new version

# EXAMPLE OF OFF-BOX SCRIPTING

- I have 100 devices on the network.
- I have to update all of my devices with a new snmp community
- Think like a programmer:
  - Use Python on a server/laptop as the point-of-control to remotely manage Junos devices.
  - Steps:
    - Find/Create a list of all the devices IPs and Credentials
    - For every device in the list
      - Connect to device
      - load the configuration change
      - commit the configuration change
      - Close the session
    - Repeat with the next device until we finish the list

# WHEN USING AUTOMATION

- The Build phase
  - Around the initial design and installation of a network component.
  - ZTP, netconify, Python, Openclos, Ansible can help in building network
- The Configure phase
  - Deploy on demand configuration and software changes to the platform.
  - PyEZ, Ansible, Puppet, Chef can help in configuring or reconfiguring the network for new services
- The Audit phase
  - Deals with automating the process of monitoring operational state of the platform
  - PyEZ, JSNAPy, ANSIBLE, OPEN-NTI, JUNOS REST API, SNMP can be used to help monitoring or auditing the network

# PYTHON

# WHAT IS PYTHON?

- A programming language
- Popular. Widely used.
- Contribution from a very large community
  - Lots of modules available extending the capabilities of the language (repository https://pypi.python.org/pypi)
- Easy to learn
- Indentation matters
- Versions:
  - Python 2.7 is still mostly in use.
    - PyEZ uses Python 2.7
    - Ansible uses Python 2.7
  - Python 3.5 adoption is coming.
    - Python 3.X adoption inhibited because of the many community modules that require 2.7

# PIP (package manager/installer program)

- pip stand for "Pip Installs Packages" or "Pip Installs Python".
- pip is a package management system used to find, install and manage Python packages.
- Many packages can be found in the Python Package Index (PyPI).
  - This is a repository for Python.
  - There are currently 86000 packages.
  - https://pypi.python.org/pypi.
- You can use pip to find packages in Python Package Index (PyPI) and to install them.

# PIP (package manager/installer program)

- pip --help to understand the various options

```
ksator@ubuntu:~$ pip --help
```

- pip list to list  installed packages.
  - ncclient (netconf client), Paramiko (ssh client), junos-eznc (PyEZ), pip ☺, jinja2 to manage templates, netaddr to manage ip addresses, requests to handle REST calls, …

```
ksator@ubuntu:~$ pip list
```

- pip search to searches packages related to any topic (contrail or Juniper or vpn or regex …).
- Other pip options frequently used are install and uninstall ...

# USING PYTHON

- To execute a python script, type the command python followed by the path to the .py script

```
ksator@ubuntu:~$ python pytraining/facts/print_facts.py
```

- To read a python script, use an advanced text editor like sublime text as example

- To start the python interpreter shell, type the command python.

```
ksator@ubuntu:~$ python
>>>
```

- To exit python interpreter shell, type quit() or exit() or use ctrl+d

```
>>> quit()
ksator@ubuntu:~$
```

# VARIABLES

- Variables store values
- Declare the name of the variable and use the assignment operator = to assign the value
- We do not declare the variable type. The value assigned to the variable determines the type
  - Integer, Floats, String, Lists, Dictionaries, and more types.
  - Dynamically typed

# INTEGERS

```
>>> # this is a comment
>>> a = 192 # use the assignment operator = to assign a value to a variable
>>> a
192
>>> type(a) # Integers are whole numbers
<type 'int'>
>>> b = 10
>>> a+b # Manipulate integers with arithmetic operators
202
>>> b/3
3
```

# COMPARAISON OPERATORS

- Comparison operators compare two values and return a Boolean

```
>>> a
192
>>> b
10
>>> a==b
False
>>> a!=b
True
>>> a>b
True
>>> a<=b
False
```

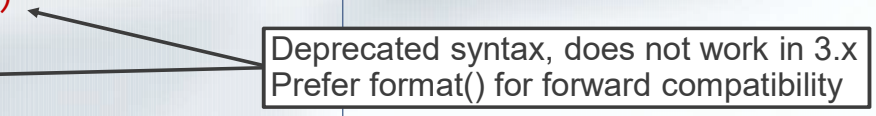
# STRINGS

- Use single quotes ' or double quotes ".
- To create a multi lines string, use three single quotes or three double quotes

```
>>> banner="you are accessing a restricted system"
>>> type(banner)
<type 'str'>
>>> banner
'you are accessing a restricted system'
>>> print banner
you are accessing a restricted system
>>> dir(str) # use dir to get the list of available functions for strings
[ ... 'upper', ...]
>>> help(banner.upper) # use help to get help with a function.
Help on built-in function upper:
upper(...)
    S.upper() -> string
    Return a copy of the string S converted to uppercase.
>>> print banner.upper()
YOU ARE ACCESSING A RESTRICTED SYSTEM
```

# PRINT

```
>>> hostname="ex4200-1"
>>> ip="172.30.179.101"
>>>
>>> # use + to concatenate strings
>>> print "the device hostname is " + hostname
the device hostname is ex4200-1
>>> print "the device " + hostname + " has the ip " + ip
the device ex4200-1 has the ip 172.30.179.101
>>>
>>> print "the device %s has the ip %s" % (hostname, ip)
the device ex4200-1 has the ip 172.30.179.101
>>> print "%s %s" % (hostname,ip)
ex4200-1 172.30.179.101
>>>
>>> print "the device {0} has the ip {1}".format(hostname, ip)
the device ex4200-1 has the ip 172.30.179.101
>>> help (str.format)
```



# LISTS

- A collection of items
- Items are ordered
- Items separated by commas
- Items are enclosed within square brackets [ ]
- A list is iterable: a “for loop” iterates over its items

# LISTS

```
>>> my_devices_list=["172.30.108.11", "172.30.108.14", "172.30.108.141"]
>>> type (my_devices_list)
<type 'list'>
>>> my_devices_list [0] # access to an item
'172.30.108.11‘
>>> my_devices_list[-1]
'172.30.108.141'
>>> # Check if an item is a list with membership operators
>>> "172.30.108.14" in my_devices_list
True
>>> dir(my_devices_list) # get the available functions for lists
[... 'append', 'count', 'index', 'insert', 'pop', 'remove', 'sort‘  …]
>>> help(my_devices_list.insert) # to get help with the insert function
>>> my_devices_list.insert(1,'172.30.108.176')
>>> my_devices_list
['172.30.108.11', '172.30.108.176', '172.30.108.14', '172.30.108.141']
```

# DICTIONARIES

- Collection of key-value pairs
- We use dictionaries to associate values to keys
- Keys are unique
- Items are unordered
- Use curly { } brackets to declare the dictionary.
  - Separate the key and value with colons
  - Use commas between each pair
- Use square brackets [ ] to retrieve the value for a key
- A dictionary is iterable: a “for loop” iterates over its keys.

# DICTIONARIES

```
>>> this_is_a_dictionary={'domain': 'jnpr.net', 'hostname': 'EX-Backbone', "time_zone": 'Europe/Paris'}
>>> this_is_a_dictionary
{'domain': 'jnpr.net', 'hostname': 'EX-Backbone', 'time_zone': 'Europe/Paris'}
>>>type (this_is_a_dictionary)
<type 'dict'>
>>> this_is_a_dictionary.keys()
['domain', 'hostname', 'time_zone']
>>> this_is_a_dictionary.values()
['jnpr.net', 'EX-Backbone', 'Europe/Paris']
>>> this_is_a_dictionary["hostname"] # Query a dictionary
'EX-Backbone
>>> this_is_a_dictionary["ntp_server"]="172.17.28.5" # Add an item to a dictionary
>>> this_is_a_dictionary
{'ntp_server': '172.17.28.5', 'domain': 'jnpr.net', 'hostname': 'EX-Backbone', 'time_zone': 'Europe/Paris'}
```

# PRINT WITH PPRINT

```
>>> this_is_a_dictionary
{'ntp_server': '172.17.28.5', 'domain': 'jnpr.net', 'hostname': 'EX-Backbone', 'time_zone': 'Europe/Paris'}
>>> print this_is_a_dictionary
{'ntp_server': '172.17.28.5', 'domain': 'jnpr.net', 'hostname': 'EX-Backbone', 'time_zone': 'Europe/Paris'}
>>> # Print a dictionary with pprint (pretty print). pprint is not a built-in function. it is provided by
the module pprint.
>>> from pprint import pprint
>>> pprint (this_is_a_dictionary)
{'domain': 'jnpr.net',
 'hostname': 'EX-Backbone',
 'time_zone': 'Europe/Paris',
 'ntp_server': '172.17.28.5' }
>>> from pprint import pprint as pp
>>> pp (this_is_a_dictionary)
{'domain': 'jnpr.net',
 'hostname': 'EX-Backbone',
 'time_zone': 'Europe/Paris',
 'ntp_server': '172.17.28.5' }
```

# FOR LOOPS

- Use “for loops” if you have something you need to iterate
  - with a list, it iterates over its items
  - with a dictionary, it iterates over its keys
  - with a file, it iterates over its lines
  - …
- Easy syntax
  - Indentation matters: use spaces for the indented block of statements

```
for (expression):
␣statement(s)
```

- For each iteration of the iterable, the “for loop” executes the statements in the loop body

# FOR LOOPS WITH A LIST

- If we use a for loop with a list, it iterates over its items.

```
>>> my_devices_list
['172.30.108.11', '172.30.108.133', '172.30.108.133', '172.30.108.14',
'172.30.108.176', '172.30.108.254']
>>> for device in my_devices_list:
      print ("the current device is: " + device)



the current device is: 172.30.108.11
the current device is: 172.30.108.133
the current device is: 172.30.108.133
the current device is: 172.30.108.14
the current device is: 172.30.108.176
the current device is: 172.30.108.254
>>>
```

# CONDITIONALS: IF…ELIF…ELSE

- Syntax:
  - Indentation matters: use spaces for the indented blocks of statements

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
else:
    statement(s)
```

- Executes some statements if an expression evaluates to true
- Elif and Else are optional
- Elif means “else if”
- Else specify actions to take if no condition was met previously.

# CONDITIONALS: IF…ELIF…ELSE

- Use a comparison operator in the if or elif expression

```
>>> my_devices_list
['172.30.108.11', '172.30.108.133', '172.30.108.133', '172.30.108.14',    '172.30.108.176',
'172.30.108.254']
>>> for device in my_devices_list:
     if device=='172.30.108.14':
      print "172.30.108.14 was found in my_devices_list"


172.30.108.14 was found in my_devices_list
>>>
```

# PYTHON BUILDING BLOCKS

- Module:
  - A file with Python code. A python file.
  - The file name is the module name with the suffix .py appended (module.py).
  - A module can define functions, classes, variables ...
- Package: several python modules all together in a directory, accompanied with a file named __init__.py. The file __init__.py can be empty.

# PYTHON BUILDING BLOCKS

- Function:
  - A function returns a value. Call a function passing arguments.
  - There are many built-in functions. You can also create your own functions.
  - A function is defined once and can be called multiple times.

# PYTHON BUILDING BLOCKS

- Class:
  - Classes define objects.
  - Call a class passing arguments. The returned value is an object. So each instance of a class is an object.
- Method:
  - A class defines functions available for this object (in a class, these functions are called methods)
  - A method is a function defined in a class.
  - To call a method, we first need to create an instance of the class. Once you have an instance of a class, you can call a method for this object.

# MODULES FOR NETWORK ENGINEERS

- Python allows you to import modules to reuse code.
  - Good programmers write good code; great programmers reuse/steal code ☺
  - Importing a module is done without using the .py extension
- Anyone can create modules for private uses or to share with community
- Some very nice Python modules/packages for network engineers:
  - netaddr: a Python library for representing and manipulating network addresses
  - re: regular expressions
  - requests: rest api manipulation
  - jinja2: generate documents based on templates
  - Yaml: “users to programs” communication (to define variables)
  - PyEZ: Python library to interact with Junos devices

# MANIPULATE IP ADDRESSES WITH PYTHON

# PYTHON NETADDR PACKAGE

- There are many Python modules to manipulate IP addresses: ipaddr (google contribution), ipaddress (easy but requires python 3), IPy, netaddr, …
- netaddr is a Python package to manipulate IP addresses and subnets.
  - IPAddress is a class in module netaddr.ip. An IPAddress instance is an individual IPv4 or IPv6 address object (without net mask)
  - IPNetwork is a class in module netaddr.ip. An IPNetwork instance is an IPv4 or IPv6 network or subnet object

# THE CLASS IPADDRESS

- Import the class IPAddress

```
>>> from netaddr import IPAddress
```

- Instantiate the class IPAddress

To instantiate a class, declare a variable and call the class passing arguments. This assigns the returned value (the newly created object) to the variable.

```
>>> ip=IPAddress('192.0.2.1')
>>> type(ip)
<class 'netaddr.ip.IPAddress'>
>>> ip
IPAddress('192.0.2.1')
>>> print ip
192.0.2.1
```

- Then you can easily play with the created objects using methods and properties.

# THE CLASS IPADDRESS

- Some methods and properties:

```
>>> ip
IPAddress('192.0.2.1')
>>> ip.version
4
>>> ip.is_private()
False
>>> ip.is_unicast()
True
>>> ip.is_multicast()
False
>>> ip.bits()
'11000000.00000000.00000010.00000001‘
>>>
>>>
>>> ip
IPAddress('192.0.2.1')
>>> ip+1
IPAddress('192.0.2.2')
>>> ip+255
IPAddress('192.0.3.0')
```

# THE CLASS IPNEWORK

- The class IPNetwork is define in the package netaddr.ip
- Each instance of the class IPNetwork is an object (a subnet)
- Once you have created an instance of the class IPNetwork, you can use the methods defined in the class IPNetwork with this subnet.

# THE CLASS IPNEWORK

- Import the class IPNetwork

```
>>> from netaddr import IPNetwork
```

- Instantiate the class IPNetwork

To instantiate a class, declare a variable and call the class passing arguments. This assigns the returned value (the newly created object) to the variable.

```
>>> net=IPNetwork('192.0.2.0/24')
>>> type(net)
<class 'netaddr.ip.IPNetwork'>
>>> net
IPNetwork('192.0.2.0/24')
>>> print net
192.0.2.0/24
>>>
>>> net[0]
IPAddress('192.0.2.0')
>>> net[-1]
IPAddress('192.0.2.255')
>>> print net[-1]
192.0.2.255
```

# THE CLASS IPNETWORK

- Some properties:

```
>>> net.version
4
>>> net.netmask
IPAddress('255.255.255.0')
>>> net.hostmask
IPAddress('0.0.0.255')
>>> net.network
IPAddress('192.0.2.0')
>>> net.broadcast
IPAddress('192.0.2.255')
>>> net.size
256
>>> net.prefixlen
24
```

- Some methods:

```
>>> net.is_unicast()
True
>>> net.is_private()
False
>>> net.is_reserved()
False
>>> net.next(12)
IPNetwork('192.0.14.0/24')
>>> net.previous()
IPNetwork('192.0.1.0/24')
```

# MANIPULATE IP ADDRESSES

- Test if an IP address belongs to a subnet. Use the membership operator (in, not in)

```
>>> from netaddr import IPNetwork, IPAddress
>>> net=IPNetwork('192.0.2.0/24')
>>> ip=IPAddress('192.0.2.1')
>>> ip in net
True
```

# MANIPULATE IP ADDRESSES

- Generates the IP addresses for a subnet

```
>>> from netaddr import IPNetwork
>>> net=IPNetwork('192.0.2.0/29')
>>> for ip in net:
     print ip

192.0.2.0
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
192.0.2.5
192.0.2.6
192.0.2.7
```

# MANIPULATE IP ADDRESSES

- The method iter_hosts provides all the IP addresses that can be assigned to hosts within a subnet: for IPv4, the network and broadcast addresses are always excluded.

```
>>> from netaddr import IPNetwork
>>> net=IPNetwork('192.0.2.0/29')
>>> help(net.iter_hosts)
>>> for ip in net.iter_hosts():
    print ip
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
192.0.2.5
192.0.2.6
```

# MANIPULATE FILES WITH PYTHON

# READ A FILE

- The method read reads an open file and returns a string.
- If there is no argument, read until EOF is reached.

```
>>> f=open("python_basics/list_of_ip.txt","r")
>>> f
<open file 'python_basics/list_of_ip.txt', mode 'r' at 0x000000000317AE40>
>>> help(f.read)
>>> s=f.read()
>>> type(s)
<type 'str'>
>>> s
'172.30.179.101\n172.30.179.102\n172.30.179.103\n172.30.179.104\n172.30.179.105\n'
>>> print s
172.30.179.101
172.30.179.102
172.30.179.103
172.30.179.104
172.30.179.105
>>> f.close()
```

# WRITE CONTENT ON A FILE

- To open a file with write mode, use "w".
  - If the file doesn't exist, python will create it.
  - If the file already exists, python will overwrite its content.
- To open a file with append mode, use "a".
  - The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. You can write content without overwriting the file content.
  - If the file does not exist, it creates a new file for writing.

```
>>> f=open("python_basics/list_of_ip.txt","a")
>>> f
<open file 'python_basics/list_of_ip.txt', mode 'a' at 0x000000000317AD20>
>>> help(f.write)
>>> f.write("172.30.179.106\n")
>>> f.write("172.30.179.107\n")
>>> f.close()
```

# JINJA2 TEMPLATES

# JINJA2 PACKAGE

- Jinja2 is used to generate documents based on templates.
- Jinja2 files use a .j2 file extension
- Variables are marked in the template
  - use a {{ variable-name }} syntax.
- Supports some control structures (if and for).
  - use a {% ... %} syntax.
- There is a Jinja2 Python package
- We can use Jinja2 to handle junos templates

# JINJA2

- Lets use this jinja2 file in a python program

```
More jinja2_basics/template_int_vlan.j2
set interfaces {{ interface }} unit 0 family ethernet-switching port-mode access vlan
members {{ vlan_name }}
```

```
>>> f=open("jinja2_basics/template_int_vlan.j2") # template_int_vlan.j2 is a jinja2 file
>>> s=f.read() # s is a string with the content of the file template_interf_and_vlan.j2
>>> type(s)
<type 'str'>
>>> print s
set interfaces {{ interface }} unit 0 family ethernet-switching port-mode access vlan
members {{ vlan_name }}
>>> from jinja2 import Template # import the class Template from module jinja2.environment
>>> whatever=Template(s) # template is an instance of the class Template.
>>> type (whatever)
<class 'jinja2.environment.Template'>
>>> # render is a method from class Template
>>> print whatever.render(interface="ge-0/0/2", vlan_name="v14")
set interfaces ge-0/0/2 unit 0 family ethernet-switching port-mode access vlan members v14
>>> f.close() # close the file
```

# DEFINE PYTHON LISTS AND DICTIONARIES USING YAML

# YAML

- YAML stands for "Yaml Ain't Markup Language"
- Yaml is human-readable language.
  - Less markup  than XML.
  - A superset of JSON.
- There is a Yaml package for Python
- Used for “users to programs” communication
  - For users to provide data.
  - Used to communicate with program.
  - Designed to translate to structures which are common to various languages (cross language: Python, Perl, Ruby, etc).
  - Used to define variables value.

# YAML SYNTAX

- Yaml files use a .yaml or .yml extension
- Yaml documents begin with three dashes - - -
- Comments begin with #
- Strings are unquoted
- Indentation with one or more spaces
  - never with tabulations
- Lists: one member per line.
  - Hyphen + space for each item.
- Keys are separated from values by a colon + space.

# YAML SYNTAX FOR A LIST

- device_list.yml is a yaml file.
  - This is a YAML list
  - There is one item per line
  - Hyphen + space for each new item

```
more yaml_basics/device_list.yml
---
#IPs of the devices in the lab environment
    - 172.30.179.101
    - 172.30.179.102
    - 172.30.179.103
    - 172.30.179.104
    - 172.30.179.105
```

# TRANSFORM A YAML FILE INTO A PYTHON STRUCTURE

- Open a yaml file

```
>>> f=open('yaml_basics/device_list.yml')
>>> f
<open file 'yaml_basics/device_list.yml', mode 'r' at 0x00000000044468A0>
>>> type (f)
<type 'file'>
```

- Read the file and return a string

```
>>> s=f.read()
>>> type (s)
<type 'str'>
>>> print s
---
#IPs of devices in the lab environment
    - 172.30.179.101
    - 172.30.179.102
    - 172.30.179.103
    - 172.30.179.104
    - 172.30.179.105
```

# TRANSFORM A YAML FILE INTO A PYTHON STRUCTURE

- Import the yaml package

```
>>> import yaml
```

- Use the load function to read a string and produce the corresponding Python structure

```
>>> my_vars=yaml.load (s)
```

- my_var is a Python list ! With the content of the yaml file.

```
>>> my_vars
['172.30.179.101', '172.30.179.102', '172.30.179.103',
'172.30.179.104', '172.30.179.105']
>>> type(my_vars)
<type 'list'>
```

# YAML SYNTAX FOR A DICTIONARY

- this_is_a_dictionary.yml is a yaml file.
  - This is a YAML dictionary
  - Keys are separated from values by a colon + space.
  - There are 2 keys (interfaces and vlan_name)
  - The value for the first key is a list
  - Its so easy to transform it into a Python dictionay!

```
more yaml_basics/this_is_a_dictionary.yml
---
interfaces:
    - ge-0/0/9
    - ge-0/0/10
    - ge-0/0/16
    - ge-0/0/18

vlan_name: v14
```

# DEMO:
# CREATE JUNOS CONFIGURATION FILES WITH PYTHON, JINJA2 AND YAML

# JINJA2 AND YAML

- Lets use a jinja2 template and a yaml file to build the initial junos configuration files. We can use with a ZTP setup to configure new devices (build phase).
- We need to provide to each new device (factory default configuration) at least the following:
  - -a root password (otherwise we can not commit the conf).
  - -a management ip @ and subnet, and a route (to be able to reach remotely the new device).
  - -allow ssh connection (in case we want to ssh it).
  - -enable netconf over ssh (to be able then to use PyEZ in the run and audit phases).
  - -an hostname.
- Only the hostname and the management ip @ are unique per device.
  - So only these 2 details are define as variables in the jinja2 template.
  - The yaml file define their values for each device.

# YAML

- configuration_builder/variables_build.yml is a yaml file.
  - This is a yaml list. With 3 items. Each item is a device.
  - Each item of the list is a dictionary with the device hostname and management ip @.
  - It is extremely easy to add other devices.
  - You can use another yaml structure (i.e instead of a list of dictionaries) but in that case you'll need to parse it differently from the jinja2 and python files.

```
pytraining@py-automation-master:~$ more configuration_builder/variables_build.yml
```

# JINJA2

- configuration_builder/template_build.j2 is a jinja2 template.
  - this is the template to build the initial junos configuration file.
  - It uses the variables defined in the yaml file.

```
pytraining@py-automation-master:~$ more configuration_builder/template_build.j2
```

# PYTHON

- configuration_builder/configuration_builder.py is a python script.
  - It uses the jinja2 template and the yaml file to create the initial junos configuration file for each device defined in the yaml file.
  - You can use these files with a ZTP setup to configure automatically new devices (build phase).

```
pytraining@py-automation-master:~$ more configuration_builder/configuration_builder.py
```

# BUILD A DOC BASED ON A JINJA2 TEMPLATE

- Use the python to generate the junos configuration file.

```
pytraining@py-automation-master:~$ python configuration_builder/configuration_builder.py
Start configuration building
generate config file for device qfx5100-10 : conf_file_build_phase_qfx5100-10.conf
generate config file for device qfx5100-6 : conf_file_build_phase_qfx5100-6.conf
generate config file for device qfx5100-8 : conf_file_build_phase_qfx5100-8.conf
done

pytraining@py-automation-master:~$ ls | grep build
conf_file_build_phase_qfx5100-8.conf
conf_file_build_phase_qfx5100-10.conf
conf_file_build_phase_qfx5100-6.conf

pytraining@py-automation-master:~$ more conf_file_build_phase_qfx5100-8.conf
pytraining@py-automation-master:~$ more conf_file_build_phase_qfx5100-10.conf
pytraining@py-automation-master:~$ more conf_file_build_phase_qfx5100-6.conf
```

# JUNOS AUTOMATION WITH PYEZ LIBRARY

*From CLI to Python*

# PYEZ AGENDA

- PYEZ INTRODUCTION
- CONNECT TO DEVICES,  RETRIEVE FACTS WITH PYEZ
- CONFIGURATION MANAGEMENT WITH PYEZ
- AUDIT MANAGEMENT WITH PYEZ

# PYEZ INTRODUCTION

# PYEZ

- Allows to manage Junos devices
- Not tied to Junos version or to Junos product.
- A Juniper package for Python
  - A package is a collection of Python modules
  - Provides classes and methods
- A Python framework
  - Provides code that is useful for larger applications.
  - Used by Ansible
- Current PyEZ version in 1.3.1
- Has been tested with Python 2.6 and 2.7.
  - Not supported with Python 3.x due to dependencies with other Python modules such as ncclient that do not yet support Python 3.x

# NETCONF PROTOCOL

- PyEZ uses Netconf.
  - You need to enable Netconf on your devices.
- Netconf is a Protocol to manipulate configuration of network devices
  - IETF standard (RFCs)
  - Implemented by most vendors
  - TCP transport, SSH encryption, XML encoding
  - Uses RPC (remote procedure call) over SSH
  - Client/server communication (the server is the network device)
  - Server default port must be 830 and should be configurable (RFC 6242)

# NETCONF PROTOCOL

- To enable the NETCONF service on the default port (830) on your devices

```
lab@ex4200-1# set system services netconf ssh
lab@ex4200-1# commit
```

- In order to enable NETCONF using another port, use this junos command

```
lab@ex4200-1# set system services netconf ssh port port-number
```

- You might want to create another user on your devices for PyEZ (to trace PyEZ activities)

# CONNECT TO DEVICES, RETRIEVE FACTS WITH PYEZ

# DEMO:
# CONNECT TO DEVICES AND RETRIEVE FACTS

- Let's execute this python program (print_facts.py).
- It prints the hostname and junos version for a list of devices defined into the program :

```
python facts/print_facts.py
the device ex4200-2 is a EX4200-24T running 12.3R11.2
The device ex4200-1 is a EX4200-24T running 12.2R2.4
the device ex4200-3 is a EX4200-24T running 12.3R11.2
the device ex4200-4 is a EX4200-24T running 12.3R11.2
```

- It also write the output here

```
more my_devices_inventory.txt
```

# IMPORT THE CLASS DEVICE

- Import the class Device from the package PyEZ.
- The class Device is defined in the module device (device.py) in the package jnpr.junos.
- The class Device provides methods:
  - For connecting to devices
  - For retrieving facts (such as software version, serial number, …) from the devices

```
>>> from jnpr.junos import Device
# Verify that the Device class has been loaded
>>> dir()
```

# INSTANTIATE THE CLASS DEVICE

- Instantiate the class Device by declaring a variable (a_device) and calling the class Device passing arguments (your device credentials). This assigns the returned value (the newly created object) to the variable a_device. Example for qfx5100-3

```
>>> a_device=Device (host="10.19.10.103", user="pytraining", password="Poclab123")
```

- The object a_device is an instance of the class Device

```
>>> type (a_device)
<class 'jnpr.junos.device.Device'>
>>> a_device
Device(10.19.10.103)
```

# LIST AVAILABLE METHODS AND PROPERTIES

- List the available methods and properties for the object a_device.
  - Some methods are open, close, …
  - Some properties are facts, …

```
>>> dir(Device)
```

# GET HELP WITH THE CLASS DEVICE

- Get help on the object a_device

```
>>> help(Device)
```

- Get help on a method or property of the class Device (example with the method close)

```
>>> help(Device.close)
```

# METHODS AND PROPERTIES IN THE CLASS DEVICE

- Use the method open to connect to the device

```
>>> a_device.open()
Device(10.19.10.103)
```

- To get the properties of the object a_device

```
>>> a_device.user
'pytraining'
>>> a_device.connected #check if the connection with your switch is still open
True
```

- Use the method close the connection to the device

```
>>> a_device.close()
>>> a_device.connected
False
```

# FACTS

- By default, device facts (such as software-version, serial-number, etc.) are retrieved when the connection is established.
- Facts is a property defined in the class Device. This is a dictionary. This command returns the facts.

```
>>> from jnpr.junos import Device
>>> a_device=Device (host="10.19.10.103", user="pytraining", password="Poclab123")
>>> a_device.open()
>>> a_device.connected #check if the connection with your switch is still open
True
>>> type(a_device.facts)
<type 'dict'>
```

# FACTS

- Pretty print the facts with pprint

```
>>> from pprint import pprint as pp
>>> pp (a_device.facts)
{'2RE': False,
 'HOME': '/var/home/remote',
 'RE0': {'last_reboot_reason': '0x2:watchdog ',
         'mastership_state': 'master',
         'model': 'EX4200-24T, 8 POE',
         'status': 'OK',
         'up_time': '4 days, 3 minutes, 45 seconds'},
 'domain': 'poc-nl.jnpr.net',
 'fqdn': 'ex4200-1.poc-nl.jnpr.net',
 'hostname': 'ex4200-1',
 'ifd_style': 'SWITCH',
 'master': 'RE0',
 'model': 'EX4200-24T',
 'personality': 'SWITCH',
 'serialnumber': 'BM0210118154',
 'switch_style': 'VLAN',
 'vc_capable': True,
 'vc_mode': 'Enabled',
 'version': '12.2R2.4',
 'version_RE0': '12.2R2.4',
 'version_info': junos.version_info(major=(12, 2), type=R, minor=2, build=4)}
```

# FACTS

- Select some device facts

```
>>> a_device.facts["hostname"]
'ex4200-1'
>>> a_device.facts["version"]
'12.2R2.4'
>>> a_device.facts["version"]=="14.1R1.2"
False
```

# REVIEW THE PRINT_FACTS PROGRAM

- The program prints the hostname and junos version for a list of devices defined in the program :

```
python facts/print_facts.py
the device ex4200-1 is a EX4200-24T running 12.2R2.4
the device ex4200-2 is a EX4200-24T running 12.3R11.2
the device ex4200-3 is a EX4200-24T running 12.3R11.2
the device ex4200-4 is a EX4200-24T running 12.3R11.2
```

- Have a look at output file.

```
more my_devices_inventory.txt
```

- Have a look at the program.

```
more facts/print_facts.py
```

# CONFIGURATION MANAGEMENT WITH PYEZ

# IMPORT THE CLASS CONFIG

- PyEZ provides us the necessary pieces of code to automate configuration deployment
- Import the class Config from the module config.py in the utils package

```
>>>from jnpr.junos.utils.config import Config
```

- call the dir function  without argument to get  the list of the names defined in the current scope. The class Config is now in the current scope.

```
>>>dir()
```

# METHODS DEFINED IN THE CLASS CONFIG

- List the available methods for the class Config

```
>>> dir(Config)
```

- Some methods for the class Config:
  - Load: apply changes into the candidate conf
  - Pdiff: display conf changes between active and candidate
  - Commit-check
  - Commit: commit a candidate conf
  - Rollback
  - Rescue
  - Lock: lock the candidate config
  - Unlock: unlock the candidate config

# GET HELP WITH THE CLASS CONFIG

- Get help on the class Config

```
>>> help(Config)
```

- Get help on the Config's methods.
  - Example with method lock

```
>>> help(Config.lock)
```

# INSTANTIATE THE CLASS CONFIG

- Define the candidate configuration.
  - Instantiate the class Config  by declaring a variable (cfg) and calling the class passing an argument (a_device).
  - This assigns the returned value (the newly created object) to the variable cfg.
    - cfg is the candidate configuration for the device a_device

```
>>> a_device.connected #check if the connection with your switch is still open
True
>>> cfg = Config(a_device)
>>> type (cfg)
<class 'jnpr.junos.utils.config.Config'>
```

# CHANGE THE CANDIDATE CONFIGURATION

- There are different ways to load changes to the candidate configuration. Lets see some of them here:

```
>>> cfg.load("set interfaces ge-0/0/23 description PyEZ", format='set')
<Element load-configuration-results at 0x7f77c8431ef0>

>>> #conf is a variable. It's a string.
>>> conf='''set vlans vlan-927 vlan-id 927
set vlans vlan-927 description "created with python"'''
>>> print conf
set vlans vlan-927 vlan-id 927
set vlans vlan-927 description "created with python"
>>> cfg.load(conf, format='set')
<Element load-configuration-results at 0x7f77c8431560>

>>> # confjunos.conf is a file with junos commands with the format set that define vlan 911
>>> cfg.load(path="configuration_management/confjunos.conf", format='set')
<Element load-configuration-results at 0x7f77c84317a0>
```

# COMPARE CONFIGURATIONS

- Compare the candidate configuration and the active configuration (or a provided rollback) with the method pdiff. Examples:

```
>>> cfg.pdiff()
[edit interfaces]
+   ge-0/0/23 {
+       description PyEZ;
+   }
[edit vlans]
+   vlan-911 {
+       description "created with python";
+       vlan-id 911;
+   }
+   vlan-927 {
+       description "created with python";
+       vlan-id 927;
+   }

>>> cfg.pdiff(rb_id=1)
```

# ROLLBACK THE CANDIDATE CONFIGURATION

- Rollback the candidate configuration to either the last active or a specific rollback number with the method rollback. Examples:

```
>>> cfg.rollback()
>>> cfg.rollback(rb_id=1)
```

# CONFIGURATION MANAGEMENT

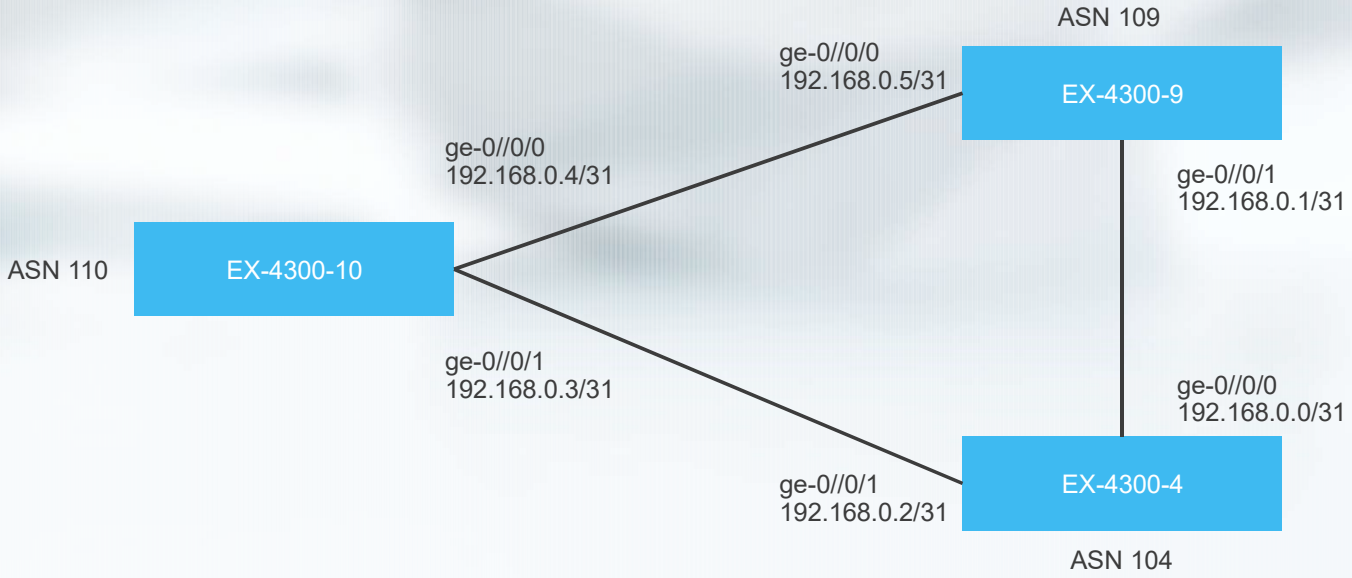
- Commit a candidate configuration with the method commit. Some examples:

```
>>> cfg.commit()
>>> cfg.commit(confirm=2)
>>> cfg.commit(comment="from pyez")
```

# DEMO: RUN PHASE
# USE AUTOMATION TO APPLY COMPLEX CONFIGURATION CHANGES ACROSS A LARGE NETWORK

# DEMO JINJA2 AND YAML

- Lets use PyEZ and Jinja2 and Yaml to apply a configuration change across a list of devices.
  - In this example we will configure some external bgp neighbors and all the required other details (bgp policies, interface configurations …)



# DEMO JINJA2 AND YAML AND PYEZ

```
pytraining@py-automation-master:~$ python configuration_builder/configuration_builder_2.py
Start configuration building
generate config file for device qfx5100-10 : conf_file_run_phase_qfx5100-10.conf
generate config file for device qfx5100-6 : conf_file_run_phase_qfx5100-6.conf
generate config file for device qfx5100-8 : conf_file_run_phase_qfx5100-8.conf
done
applying the conf to the devices ...
configuration commited on qfx5100-10
configuration commited on qfx5100-6
configuration commited on qfx5100-8
done
pytraining@py-automation-master:~$ ls | grep run
conf_file_run_phase_qfx5100-8.conf
conf_file_run_phase_qfx5100-10.conf
conf_file_run_phase_qfx5100-6.conf
pytraining@py-automation-master:~$ more conf_file_run_phase_qfx5100-8.conf
pytraining@py-automation-master:~$ more conf_file_run_phase_qfx5100-10.conf
pytraining@py-automation-master:~$ more conf_file_run_phase_qfx5100-6.conf
```

# DEMO JINJA2 AND YAML AND PYEZ

- Lets connect on the devices and double check if everything is correct
  - We will see later how to audit the network with automation instead of manually …

```
pytraining@py-automation-master:~$ ssh qfx5100-8
pytraining@py-automation-master:~$ ssh qfx5100-10
pytraining@py-automation-master:~$ ssh qfx5100-6
```

```
pytraining@qfx5100-8> show configuration | compare rollback 1
pytraining@qfx5100-8> show system commit
pytraining@qfx5100-8> show interfaces descriptions
pytraining@qfx5100-8> show lldp neighbors
pytraining@qfx5100-8> show bgp summary
pytraining@qfx5100-8> show bgp neighbor
```

# DEMO JINJA2 AND YAML AND PYEZ

- configuration_builder/variables.yml is a yaml file.
  - It defines the variables used in a jinja2 template.
  - This is a yaml list of 3 devices.
    - Each item of the list is a dictionary with some details for a device.
    - Very easy to add other devices
    - You can use another structure (i.e instead of a list of dictionaries) but in that case you'll need to parse it differently from the jinja2 file.

```
pytraining@py-automation-master:~$ more configuration_builder/variables.yml
```

# DEMO JINJA2 AND YAML AND PYEZ

- configuration_builder/template.j2 is a jinja2 template.
  - It defines BGP neighbors and other details.
  - It uses the variables defined in the yaml file.

```
pytraining@py-automation-master:~$ more configuration_builder/template.j2
```

# DEMO JINJA2 AND YAML AND PYEZ

- configuration_builder/configuration_builder_2.py is a python script.
  - It uses the jinja2 template and yaml file to create a junos configuration file for each device defined in the yaml file.
  - It then use PyEZ to connect to the list of devices, and load and commit the configuration change

```
pytraining@py-automation-master:~$ more configuration_builder/configuration_builder_2.py
```

# AUDIT MANAGEMENT WITH PYEZ

# CLI, XML, RPC

- CLI is optimized for humans. CLI is not optimized for programs (difficult to parse CLI output from a program)
- Junos supports also XML (Extensible Markup Language) representation.
- XML is not optimized for humans (too much markup). XML can be manipulated by programs.

# CLI, XML, RPC

- When you interact with a Junos device using its command-line interface, you actually interact with:

```
pytraining@ex4200-13> show version detail | match CLI
CLI release 14.1X53-D30.3 built by builder on 2015-10-02 09:52:33 UTC
```

- Then CLI passes the equivalent XML RPC to MGD

```
pytraining@ex4200-13> show version detail | match MGD
MGD release 14.1X53-D30.3 built by builder on 2015-10-02 12:38:35 UTC
```

- Then MGD get the data
- Then MGD returns the data to CLI in the form of an XML document.
- Then CLI converts back into a human readable format for display.

# CLI, XML, RPC

- To display the output of a junos CLI command in XML format, append "| display xml" option to your CLI command.
- The "| display xml rpc" option provides you the RPC to get an XML encoded response
- Example with LLDP

```
pytraining@qfx5100-6> show lldp neighbors
pytraining@qfx5100-6> show lldp neighbors | display xml
pytraining@qfx5100-6> show lldp neighbors | display xml rpc
```

# TABLES AND VIEWS

- PyEZ (the jnpr.junos.op package) allows programmatic access to junos data on the devices (so you can audit your network programmatically instead of manually)
- It uses RPCs to get the data in an XML representation
- It then parses the XML response (so you don't need to worry about this)
- It transforms the output from XML into Python data structures (tables and views, kind of list of dictionaries) that you can easily use by Python.
  - It allows the junos data to be presented using python data structures
  - No need to parse XML in your Python code
  - This enables "pythonic" access to junos data
- PyEZ uses YAML to create tables and views
  - /usr/local/lib/python2.7/dist-packages/jnpr/junos/op/ directory

# TABLES AND VIEWS: DEMO WITH LLDP

- Let me execute this Python program.
  - It uses the op package from PyEZ for LLDP.

```
python tables_and_views/lldp_neighbor_status.py

LLDP neighbors of device 172.30.179.65 (hostname is qfx5100-10):
interface me0 has this neighbor: mgmt-13
interface ge-0/0/0 has this neighbor: qfx5100-6
interface ge-0/0/1 has this neighbor: qfx5100-8

LLDP neighbors of device 172.30.179.95 (hostname is qfx5100-6):
interface ge-0/0/0 has this neighbor: qfx5100-8
interface ge-0/0/1 has this neighbor: qfx5100-10

LLDP neighbors of device 172.30.179.96 (hostname is qfx5100-8):
interface me0 has this neighbor: mgmt-13
interface ge-0/0/0 has this neighbor: qfx5100-6
interface ge-0/0/1 has this neighbor: qfx5100-10
```

# TABLES AND VIEWS: DEMO WITH LLDP

- Let me execute this Python program
  - It uses the op package from PyEZ for LLDP.
  - It asks you for the lldp neighbor you are looking for.
    - Could be a server name
  - If it finds this hostname in the lldp neighbors table of a device in the network, it prints the device name and the interface name on which this lldp neighbor is connected.

```
python tables_and_views/search_an_lldp_neighbor.py
name of the neighbor you are looking for:qfx5100-10
this neighbor is connected to the interface ge-0/0/1 of the device qfx5100-6
this neighbor is connected to the interface ge-0/0/1 of the device qfx5100-8
Done
```

# TABLES AND VIEWS: DEMO WITH BGP

- For each device in a device list, this program prints:
  - The list of its BGP neighbors
  - The status of its BGP connections

```
python tables_and_views/bgp_states.py

status of BGP neighbors of device 172.30.179.65 (hostname is qfx5100-10):
External BGP neighbor 192.168.0.1+57665 is Established (flap count is: 0)
External BGP neighbor 192.168.0.3+58699 is Established (flap count is: 0)

status of BGP neighbors of device 172.30.179.95 (hostname is qfx5100-6):
External BGP neighbor 192.168.0.0+179 is Established (flap count is: 0)
External BGP neighbor 192.168.0.4+51736 is Established (flap count is: 0)

status of BGP neighbors of device 172.30.179.96 (hostname is qfx5100-8):
External BGP neighbor 192.168.0.2+179 is Established (flap count is: 0)
External BGP neighbor 192.168.0.5+179 is Established (flap count is: 0)
```

# TABLES AND VIEWS: DEMO WITH BGP

- Let's have a look at the program:
  - It imports the class from the module jnpr.junos.op.bgp
  - It instantiates the class (passing a device as argument)
  - It use the method get to retrieve the bgp information (send rpc and get the bgp details encoded with XML)
  - PyEZ parses the XML response and build a data structure that can be used very easily by Python.
    - bgp neighbors details are presented into python data structures
    - easy to access to the bgp details, no XML parsing

```
more tables_and_views/bgp_states.py
```

- Let's have a look at the output file:

```
more bgp_states.txt
```

# JSON

# WHAT IS JSON ?

- JavaScript Object Notation
- Syntax
  - Human readable (easier than XML)
  - Easy for programs to generate and parse (this is a Python dictionary)
- Used to exchange data between applications
  - Accept: application/json
  - Content-Type: application/json
- text file with .json suffix
- JUNOS CLI output with “| display json” is also available
  - “show version | display json” as example

# A JSON EXAMPLE

- JSON representation describing a person:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": True,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  }
}
```

# DEMO: JSON WITH GOOGLE MAPS API

- google maps GUI
  - https://www.google.co.uk/maps/place/9 HaMenofim Street  Herzliya
  - For humans
- Google map API
  - http://maps.googleapis.com/maps/api/geocode/json?address=9 HaMenofim Street  Herzliya
  - gives you back some details regarding the address
  - Results in JSON format
  - Easy to parse response (Python dictionary)

# DEMO: JSON WITH GOOGLE MAPS API

- Let's use Python to send the request and parse the response

```
ksator@ubuntu:~/pytraining$ python rest_basics/google_map_api.py
which address: 9 HaMenofim Street  Herzliya
latitude is 32.1602492
longitude is 34.8081319
```

- Let's review the code

```
$ more rest_basics/google_map_api.py
```

# REST APIs

# What is REST?

- Acronym for REpresentational State Transfer
- Used for programmatic access. To exchange data.
- Not a protocol. An architectural style:
  - client server model
  - runs over HTTP
    - Identification of resources with URIs
    - CRUD operations with standard http methods (GET/POST/PUT/DELETE)
  - often json files. can be xml files.
  - Can be cacheable
  - Stateless.
    - No concept of session nor cookie.

# REST APIs ON JUNOS

- JUNOS 15.1 supports REST API to submit RPCs
  - To know the equivalent RPC of a junos show command, use “show xxx | display xml rpc”
  - Show version -> GET-SOFTWARE-INFORMATION
  - Show interfaces -> GET-INTERFACE-INFORMATION
- You can retrieve data in XML or JSON
- You can use curl or python or a graphical rest client
- The documentation is here:

https://www.juniper.net/documentation/en_US/junos15.1/information-products/pathway-pages/rest-api/rest-api.pdf

# REST APIs CONFIGURATION ON JUNOS

- REST configuration is under “system services”
  - default port is 3000
  - REST Explorer is an optional tool (GUI) for testing

```
set system services rest http
set system services rest http rest-explorer
```

# DEMO-REST CALLS TO JUNOS WITH PYTHON

- The python program rest_basics/get_mx_software_information.py uses the JUNOS REST APIs to get some details regarding the device

```
pytraining@py-automation-master:~$ python rest_basics/get_mx_software_information.py
Software version: 15.1R2.9
Host-name: mx80-17
Product name: mx80-48t
```


JUNIPer
NETWORKS
ANSIBLE

# WHAT IS ANSIBLE?

# WHAT IS ANSIBLE?

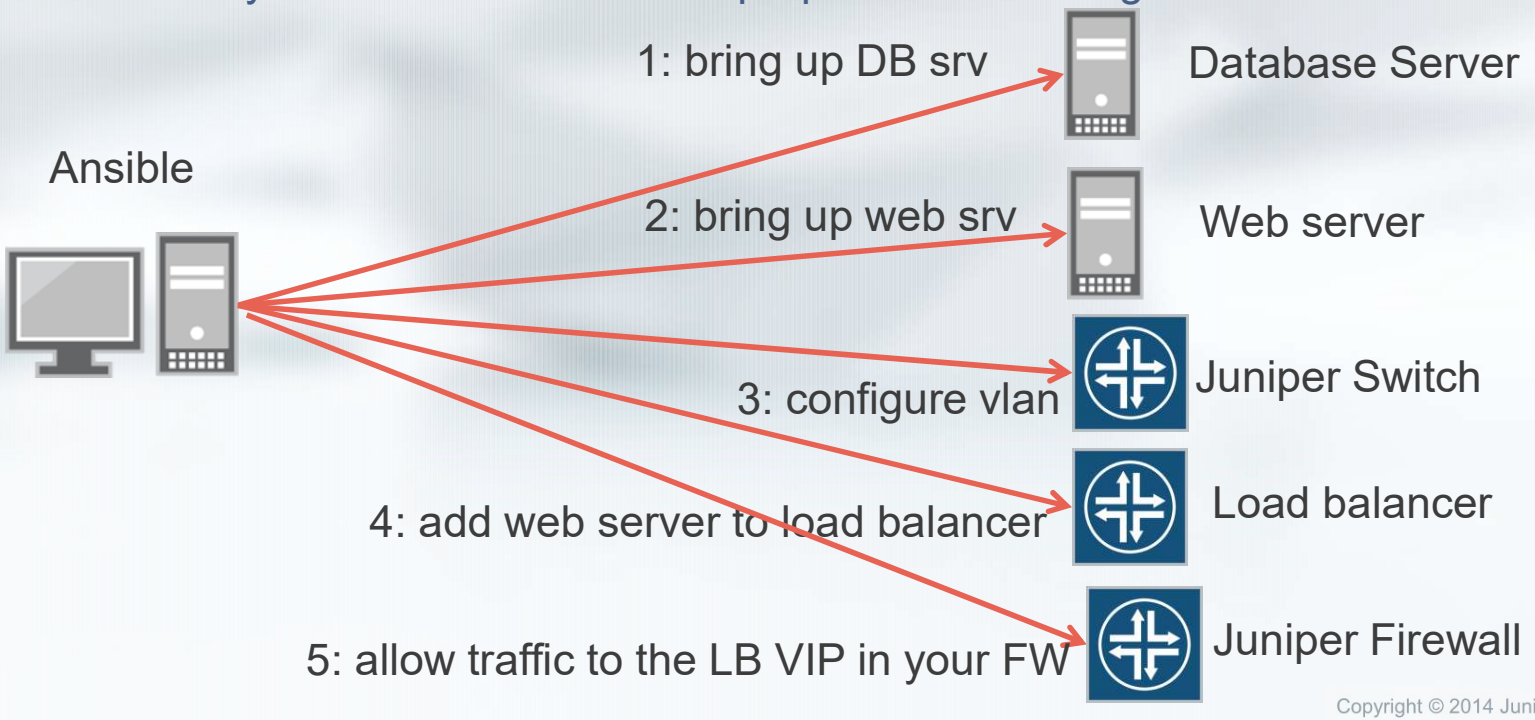
- A science fiction literature reference in the 60's ☺
- A radically simple IT automation platform
  - Ansible's goals are similar to other tools such as Puppet, Chef and Salt
  - Created by Michael DeHaan
  - Initial release in 2012
  - Current release is 2.1 (June 2016)
  - Acquired by Red Hat in 2015
    - Michael DeHaan left
  - It is very popular!
  - They hired Peter Sprygada

# WHAT CAN WE DO WITH ANSIBLE?

- This is an IT automation tool
- Initially for servers and applications
- Ansible is designed for performing actions (tasks) on multiple servers.
- Ansible uses cases are:
  - Application deployment
  - Configuration management
  - Template management
  - Orchestration
- Some nice network automation capabilities added recently.

# ORCHESTRATION WITH ANSIBLE

- You can manipulate an entire application stacks made up multiple tiers.
  - you need to bring up the database before bringing up the web servers
  - you need to take web servers out of the load balancer one at a time in order to upgrade them without downtime.
  - you need  to ensure that your networks have their proper VLANs configured



# WHY SOME PEOPLES LOVE ANSIBLE

- Agentless
- Easy to use. Human readable automation. Data, not code. Configuration in Yaml. Use Jinja2 templates.
- You can use a large library (about 500 modules) with some level of abstraction.
- Push based. No Master server. Running from everywhere (laptop).
- Networks are integral parts of IT enterprises, so Ansible has now some modules for network automation as well!
- Nice orchestration capabilities
- Has loops and conditionals
- Idempotent (but not all the modules)
- Written in Python (but no need to understand Python).
- You can add your own modules to extend Ansible capabilities.

# EASY TO USE

- Get productive quickly
  - I think it is easier for regular engineers to use Ansible versus Puppet and chef
- Easy-to-Read Syntax:
  - Human readable automation. Playbooks in Yaml.
  - Its data, not code! No special coding skills needed (you still need to learn Ansible)
  - Ansible uses the Yaml and Jinja2, so you'll need to learn some Yaml and Jinja2 to use Ansible, but both technologies are easy to pick up.
- Level of abstraction
  - Declarative approach. You describe the desirated state, indicating "what", not "how".
    - "these hosts are dbservers" or "these hosts are webservers".

# IDEMPOTENCY

- Some module are idempotent
  - Changes should only be applied when they need to be applied, and that it is better to describe the desired state of a system than the process of how to get to that state.
  - With idempotent modules, Ansible will first check the state of the device and will make a change only if the current state does not match the required end state".

# HOW IT WORKS?

# AGENT LESS

- Ansible use SSH (quite native)
  - There is no agent to install on the remote hosts
  - But some Ansible modules have requirements on the endpoints and on the Ansible server (this will be discussed later on in this presentation)
- Agent less solutions advantages:
  - when a box is not being managed by Ansible, nothing extra is running. Saves resources (memory and cpu, per vm) on endpoints.
  - more secured
  - No additional open port
  - Eliminate the need to deploy and manage agents (easier)
  - Broader application: some network devices are “closed” in a such a way 3rd party software agents cannot be loaded onto the device.
- Agent based solutions can be very nice as well (because the agent can collect many data and watch states/events on endpoints)
  - Slaltstack, Apstra, Puppet, chef.

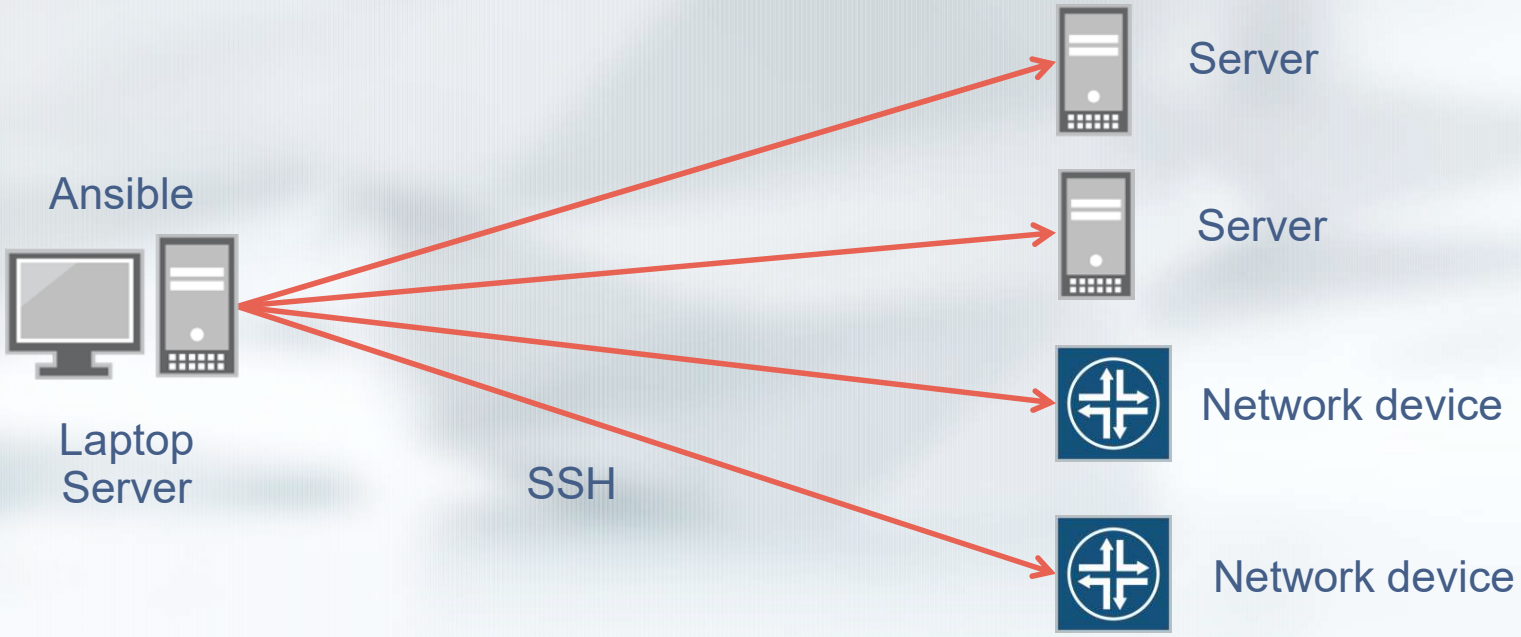
# PUSH BASED

- Ansible is push based
  - It is not pull based: where an agent installed on servers periodically check with a central service (master server) and pull configuration information form the service.
- As soon as you run the ansible-playbook command (to execute a playbook), Ansible connects to the remote servers/devices and does its thing.
- The push-based approach has a significant advantage:
  - You control when the changes happen to the servers. You don’t need to wait around for a timer to expire.

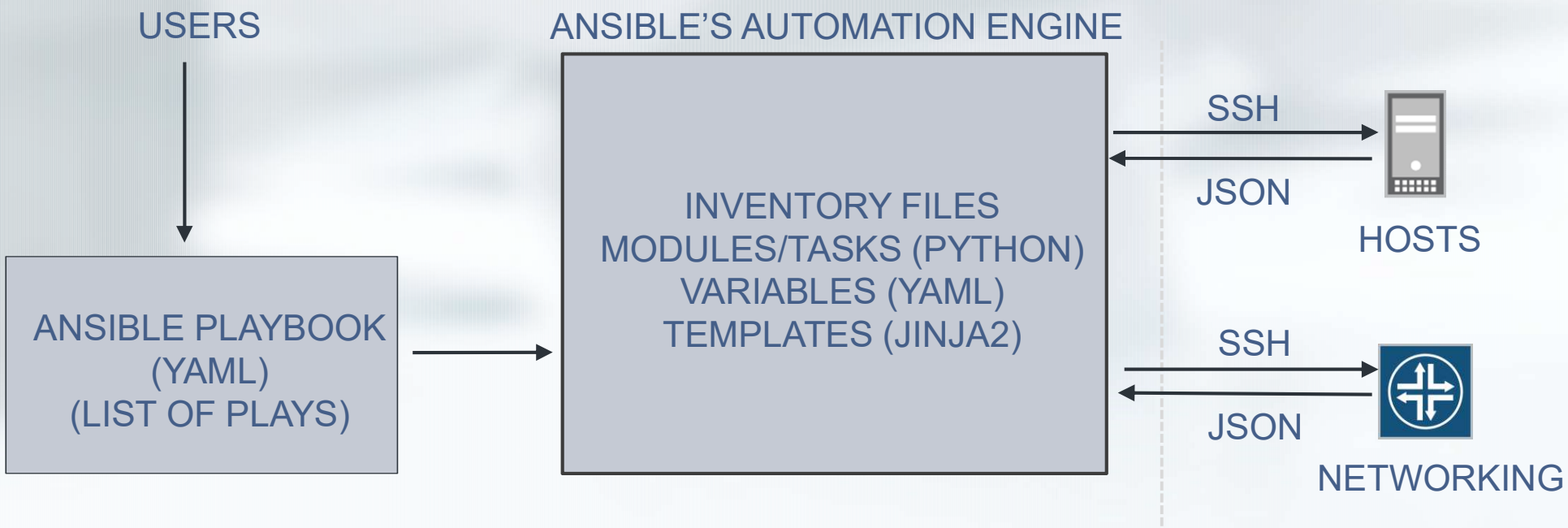
# HOW ANSIBLE WORKS?

- Standard mode uses SSH to transport python modules to the remote boxes, and then execute these python modules on remote boxes, and then eliminate them from remote hosts.
  - Python modules execution will change the device configurations.
  - Requires Python installed on remotes machines.
    - You can use the Ansible module ping to check this. It returns pong on success.
    - Some Ansible modules require also some python libraries installed on the remote hosts.
- API mode: Modules run on Ansible server. Ansible then uses a channel/API (Netconf ....) to change the device configurations.
  - “connection: local” in the playbook means the actions/tasks will be run local on the server running Ansible and not on the target host.

# HOW ANSIBLE WORKS?



# HOW ANSIBLE WORKS?



# ANSIBLE AND PYTHON

- Ansible is written in Python.
  - All Ansible core modules are written in Python
- No need to know Python to use Ansible
  - You can use ansible without understanding the python modules
  - Unless:
    - you want to write your own modules to extend Ansible capabilities.
    - You want to understand the source code
- Ansible 2.1 uses Python 2.7
  - Python 3 is not yet compatible with Ansible
- When you install Ansible, it installs automatically some Python modules:
  - paramiko (Ansible manages machines over SSH)
  - PyYAML (you interact with Ansible in yaml)
  - Jinja2 (Ansible has builtin templating capabilities)
  - httplib2 (a comprehensive HTTP client library)

# ANSIBLE AND PYTHON

- If you already know Python, Jinja2, YAML, PyEZ, JSON, Ansible is an excellent next step! It uses all of them, but with some level of abstraction.
  - All Ansible core modules are written in Python
  - Ansible use PyEZ to interact with Junos devices
  - Jinja2 is used to build documents based on templates
  - YAML is used to defines variables values
  - YAML is used to write Ansible playbooks (Ansible scripts)
  - Ansible expects modules to output JSON

# ANSIBLE KEY COMPONENTS

# ANSIBLE KEY COMPONENTS

- Inventory
- Variables
- Tasks/modules
- Plays and Playbooks

# INVENTORY

# INVENTORY FILES

```
[spine]
qfx10002-01
qfx10002-02

[leaf]
qfx5100-01
qfx5100-02
qfx5100-03
qfx5100-04
```

- text file with your inventory
  - you can have many inventory files
  - The default ansible 'hosts' file lives in /etc/ansible/hosts
- Comments begin with the '#' character
- Blank lines are ignored
- You can enter hostnames or ip addresses
- Groups of hosts (per device type, per location, ...)
- A hostname/ip can be a member of multiple groups
- Example https://github.com/dgjnpr/ansible-template-for-junos/blob/master/hosts

```
[all:children]
spine
leaf

[spine]
qfx10002-01    junos_host=192.168.0.1
qfx10002-02    junos_host=192.168.0.2

[leaf:children]
rack01
rack02

[rack01]
qfx5100-01     junos_host=192.168.0.3
qfx5100-02     junos_host=192.168.0.4

[qfx10000]
qfx10002-01
qfx10002-02

[qfx5100]
qfx5100-01
qfx5100-02

[siteA]
qfx10002-01
qfx5100-01
```

- Simple but very powerful and important
- One device can be part of multiple groups
- Can have a hierarchy of groups
- Can groups devices by type/location/roles etc …
- qfx5100-01 is part of groups:
  - All
  - leaf
  - rack01
  - qfx5100
  - siteA
- Inventory files can take advantage enumerations
  - QFX5100-[1:13]
  - host[a:z]
  - 172.16.0.[1:254]
- Inventory files can take advantage of variables

# VARIABLES

# VARIABLE FILES

```
global:
    root_hash: $1$ZUlES4dp$OUwWo1g7cLoV/aMWpHUnC/
    time_zone: America/Los_Angeles
    name_servers:
    - 192.168.5.68
    - 192.168.60.131
    ntp_servers:
    - 172.17.28.5
    snmp:
        location: "Site 1"
        contact: John Doe
        polling:
        - community: public
    routes:
        default: 10.94.194.254
```

- Variable definitions is very flexible
- A variable file is a YAML file
- Variable files can live in many places
- Variables can be defined in playbooks,
- Variables can be defined in the inventory files.
- Variable files can also be referred in a playbook (with the include_vars module)
- Accessible from templates and playbooks
- The module setup is automatically called by playbooks to gather facts (variables discovered, not set by the user) about remote hosts (setup module is not valid for junos devices so use gather_facts: no in your play)

# VARIABLES ASSOCIATE WITH GROUPS AND HOSTS

```
|-- group_vars
|   |-- leaf01
|   |   |-- file1.yaml
|   |   `-- file2.yml
|   |-- leaf02
|   |   |-- file1.yaml
|   |   `-- file2.yml
|   |-- spine
|   |   |-- file1.yaml
|   |   `-- file2.yml
|   |-- all.yaml
|   |-- qfx10000.yaml
|   `-- qfx5100.yaml
|-- host_vars
|   |-- qfx10002-01
|   |   |-- file1.yml
|   |   `-- file2.yaml
|   |-- qfx5100-01
|   |   |-- file1.yml
|   |   `-- file2.yaml
[…]
|   |-- qfx5100-04
|   |   |-- file1.yml
|   |   `-- file2.yaml
|   `-- qfx5100-05.yml
|   `-- qfx5100-06.yml
```

- Variable files can live in many places
  - host_vars folder has the variables per host
  - group_vars folder has the variables per group (qfx, spines, site1, all, …)
  - vars subfolders of roles can be used to store variables for roles
- One or multiple variable files per
  - Host
  - Group
- Very flexible

# GENERATE C

**host_vars/qfx5100-01.yaml**

```
ospf:
     interfaces:
          ge-0/0/0:
               ip: 192.168.0.1
               mask: 24
          ge-0/0/1:
               ip: 192.168.1.1
               mask: 24
Host:
     mgmt:
          ip: 10.10.10.1
          mask: 24
```

**group_vars/qfx5100.yaml**

```
Mgmt_interface: em0
```

**group_vars/all.yaml**

```
Global:
     root_hash: $1$dvewqcsarwrgvwr
Host:
     mgmt:
          mask: 24
```

```
system {
     host-name {{ inventory_hostname }};
      root-authentication {
          encrypted-password "{{ global.roo
      }
}
interfaces {
   {{ mgmt_interface }} {
      unit 0 {
          family inet {
               address {{ host.mgmt.ip }}/{
          }
      }
   }
{% for interface in ospf.interfaces %}
     {{ interface }} {
        unit 0 {
          family inet {
               address {{interface.ip }}/{{
          }
        }
     }
{% endfor %}
}

protocols {
     ospf {
          area 0.0.0.0 {
{% for interface in ospf.interfaces %}
             interface {{ interface }}
{% endfor %}
}
```

**Final version**

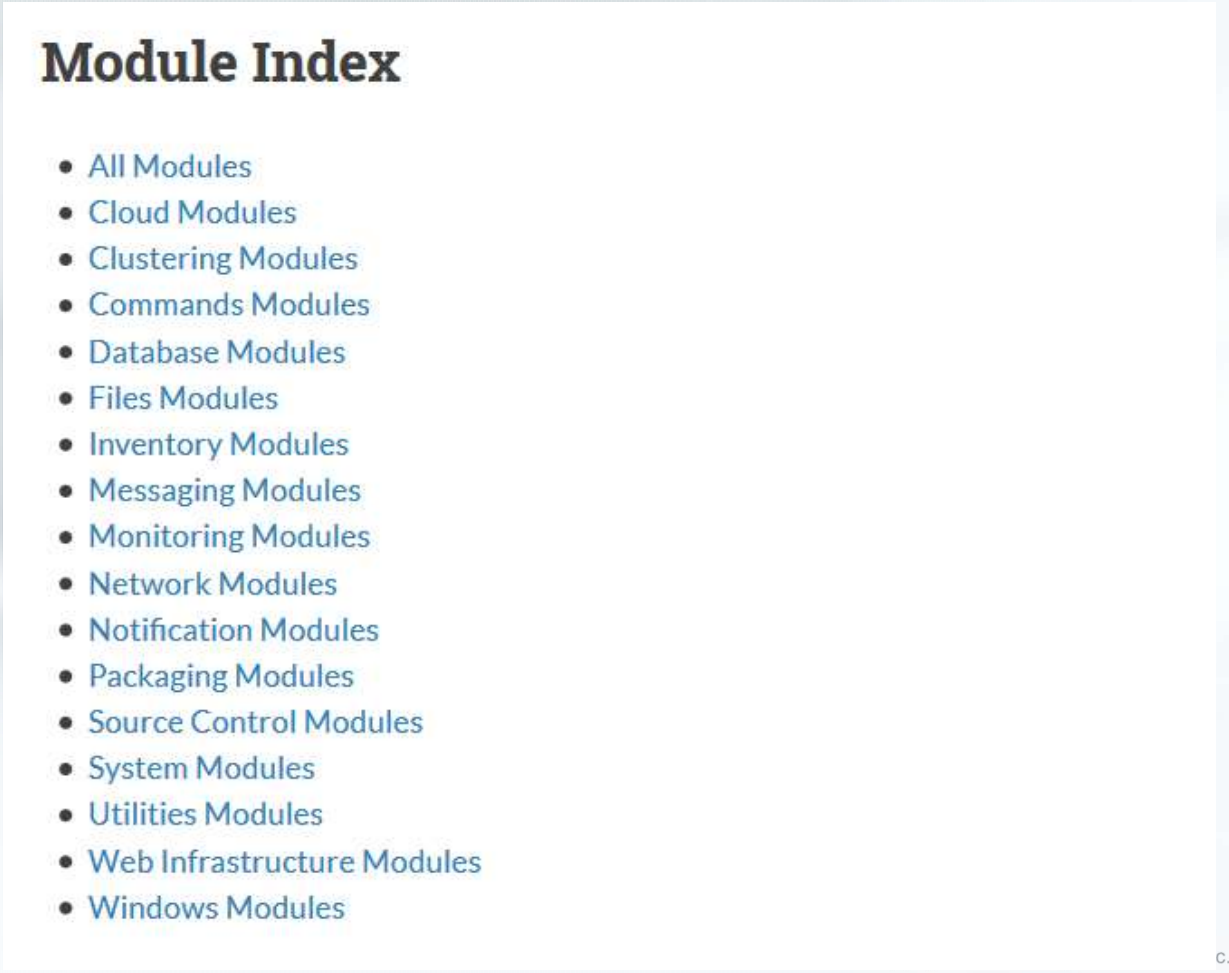
```
system {
     host-name qfx5100-01;
      root-authentication {
          encrypted-password "$1$dvewqcsarwrgvwr";
      }
}
interfaces {
   em0 {
      unit 0 {
          family inet {
               address 10.10.10.1/24;
          }
      }
   }
   ge-0/0/0{
        unit 0 {
          family inet {
               address 192.168.0.1/24
          }
        }
     }
   ge-0/0/1{
        unit 0 {
          family inet {
               address 192.168.1.1/24
          }
        }
    }
}

protocols {
     ospf {
          area 0.0.0.0 {
               interface ge-0/0/0
                interface ge-0/0/1
}
```

# MODULES

# MODULES

- Lot of modules available (Batteries included: Ansible ships with about 500 modules)



# MODULES

- http://docs.ansible.com/ansible/list_of_all_modules.html
- Some nice core modules:
  - template: Generates a file from a template and copies it to the hosts.
  - Assemble: Assembles a configuration file from fragments.
  - file: creates/removes files and directories.
  - copy: Copies files to remote hosts.
  - pip: installs python packages  (pip is required on remote hosts)
  - apt: manages packages (using apt)
  - yum: Manages packages (with yum)
  - git: deploys files from a git repo (git is required on remote hosts)
  - service: control services on remote hosts
  - pause: Pauses playbook execution
  - uri: interacts with web services

# Ansible librairies to interact with Junos

- There are two Ansible librairies to interact with Junos.
  - An Ansible library for Junos built by Juniper (hosted on the Ansible Galaxy website)
  - An Ansible library for Junos built by Ansible (since Ansible 2.1)
- You can use both of the Ansible libraries for Junos

# SOME ANSIBLE USE CASES FOR JUNOS DEVICES

- Generate configuration
- Deploy configuration
- Rollback configuration
- Retrieve configuration and facts
- Upgrade devices
- Audit the devices

# Ansible modules for Junos built by Juniper

- Hosted on the Ansible Galaxy website (https://galaxy.ansible.com/Juniper/junos/)
  - Version: 1.3.1 (June 2016)
  - Documentation: http://junos-ansible-modules.readthedocs.io/en/1.3.1/
  - Installation: Execute the “ansible-galaxy install Juniper.junos” command to download it
  - Source code https://github.com/Juniper/ansible-junos-stdlib
  - These modules use PyEZ
- Overview of modules:
  - junos_cli - Execute CLI on device and save the output locally
  - junos_commit - Execute commit on device
  - junos_get_config - Retrieve configuration of device
  - junos_get_facts - Retrieve facts for a device running Junos OS.
  - junos_install_config - Load a configuration file or snippet onto a device running Junos OS.
  - junos_install_os - Install a Junos OS image.
  - junos_rollback - Rollback configuration of device
  - junos_rpc - run given rpc
  - junos_shutdown - Shut down or reboot a device running Junos OS.
  - junos_srx_cluster - Create an srx chassis cluster for cluster capable srx running Junos OS.
  - junos_zeroize - Erase all data, including configuration and log files, on a device running Junos OS.

# Ansible core modules for Junos built by Ansible:

- Documentation: http://docs.ansible.com/ansible/list_of_network_modules.html
- Installation: They are core modules.
  - They ship with ansible itself (from Ansible 2.1).
  - Ansible 2.1 or above is required.
- Source code: https://github.com/ansible/ansible-modules-core/tree/devel/network/junos
- Modules overview :
  - junos_command - Execute arbitrary commands on a remote device running Junos
  - junos_config - Manage configuration on remote devices running Junos
  - junos_facts - Collect facts from remote device running Junos
  - junos_netconf - Configures the Junos Netconf system service
  - junos_package - Installs packages on remote devices running Junos
  - junos_template - Manage configuration on remote devices running Junos
- These modules use PyEZ (except junos_netconf)
- Some are idempotent (junos_config, junos_template, junos_netconf, junos_package)

# TASKS

- a task is an Ansible module written in python with some arguments http://docs.ansible.com/ansible/list_of_all_modules.html
- each task has a name.
- The task's name is optional, but recommended
  - for troubleshooting (Ansible print out the name of a task when it runs)
  - Names are useful when somebody else is trying to understand your playbook (including yourself in few months)
  - you can reference the name of a task with the ansible-playbook conmmand and the --start-at-task <task name> option to start the playbook at the task matching this name
- Every task must contain a key with the name of a module and a value with the arguments to that module.

# EXAMPLES OF TASKS WITH YUM MODULE

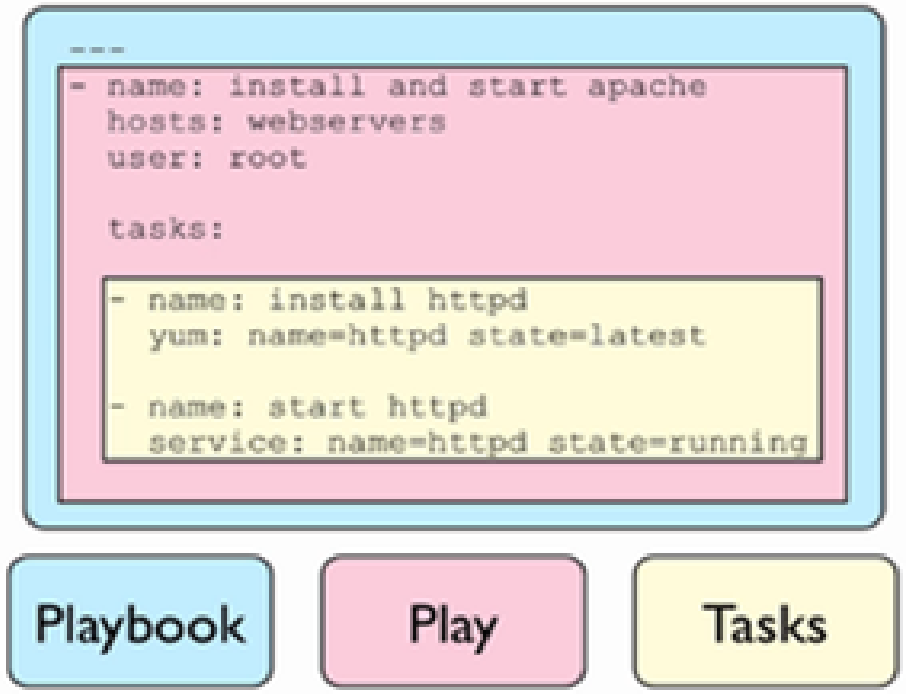
```
- name: install the latest version of Apache
  yum: name=httpd state=latest

- name: remove the Apache package
  yum: name=httpd state=absent

- name: install one specific version of Apache
  yum: name=httpd-2.2.29-1.4.amzn1 state=present
```

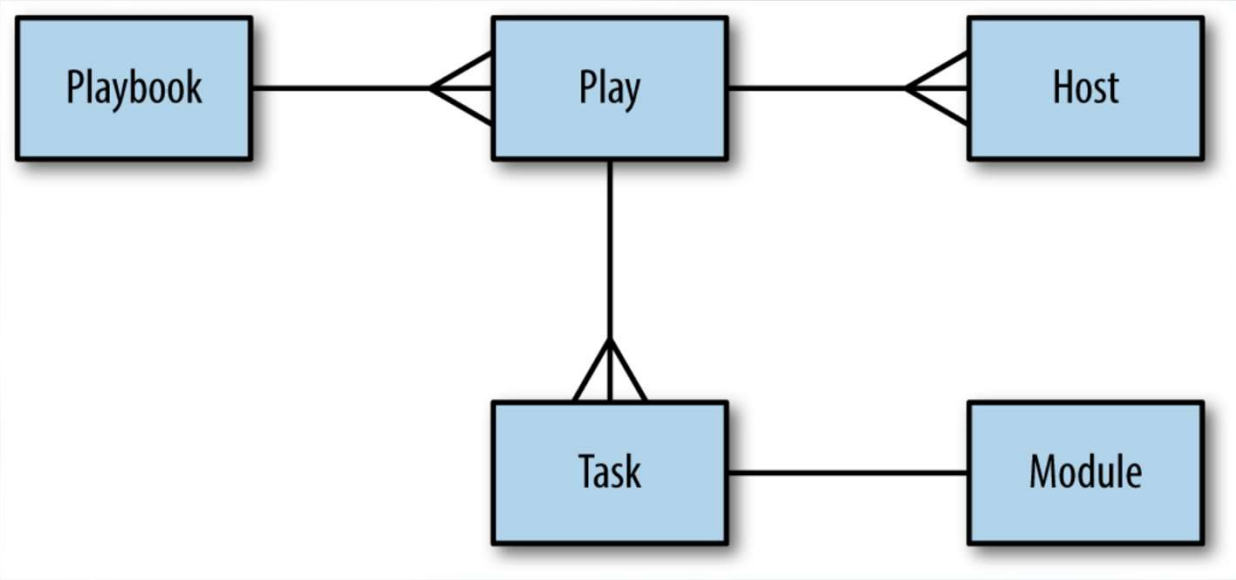
# PLAYBOOK

# A Playbook



# ANATOMY OF A PLAYBOOK

- Relationship between playbooks, plays, hosts, tasks, and modules



# PLAYBOOK

- Ansible scripts are called playbooks.
- Playbooks are written in YAML (easy for humans to read and write)
- a playbook is a YAML list of plays.
- a play has a list of tasks.
- it is a list of dictionaries
  - a playbook is a YAML list of plays
  - sometimes there is a single play
  - a play maps a selection of hosts to a list of tasks.
- playbook runs top to bottom
- you execute a playbook with ansible-playbook command
  - you can use the option -i to specify inventory host file (default=/etc/ansible/hosts)

# PLAYS

- each play has a name (key/value pair).
  - the name is displayed when the playbook is run.
- a play maps a selection of hosts to a list of tasks.
- a play has a list of tasks.
  - a task is a module written in python
- Every play must contain:
  - A set of hosts
  - A list of tasks to be executed on those hosts
- Tasks are executed in order, one at a time, against the selection of hosts, before moving on to the next task.
- Tasks can be
  - Merge files
  - Render Jinja2 Template
  - Push configuration on Junos
  - …
- a play can map a group of hosts to roles.
  - access-switch
  - Spines
  - underlay-ebgp
  - common
  - ...

# PLAYBOOK

```
---
- name: Get Junos facts
  hosts: leaves
  roles:
   - Juniper.junos
  connection: local
  gather_facts: no

  vars_prompt:
   - name: DEVICE_PASSWORD
     prompt: Device password
     private: yes

  tasks:

  - name: Retrieve information from devices running Junos
    junos_get_facts:
     host={{ inventory_hostname }}
     user=pytraining
     passwd={{ DEVICE_PASSWORD }}
     savedir=inventory
    register: junos

  - name: Print some facts when devices are not running junos 12.3R11.2
    debug: msg="device {{junos["facts"]["hostname"]}} runs version {{junos.facts.version}}"
    when: junos.facts.version != "12.3R11.2"
```

This playbook has one play (“Get Junos facts”).

This play executes two tasks on all the hosts from the group “leaves”.

The first task execute the module “junos_get_facts” with some arguments.

The second task is a conditional task (“when”) and execute the module “debug”.

# ANSIBLE JARGON

# ANSIBLE JARGON

- Playbook: Ansible scripts. Written in Yaml. A list of Plays.
- Plays: contains a set of hosts, and a list of tasks to be executed on those hosts
- Inventory file: your hosts and groups. Can be dynamic
- Variables: there are various options to define variables.
- Task: a Python/Ansible module with some arguments.
- Handlers: a triggered task
- Server facts: Like puppet. Automatically gather by setup module.
- Roles: a reusable automation content you can assign to hosts
- Galaxy: repository for Ansible roles like the ones for Junos built by Juniper.

# ROLE

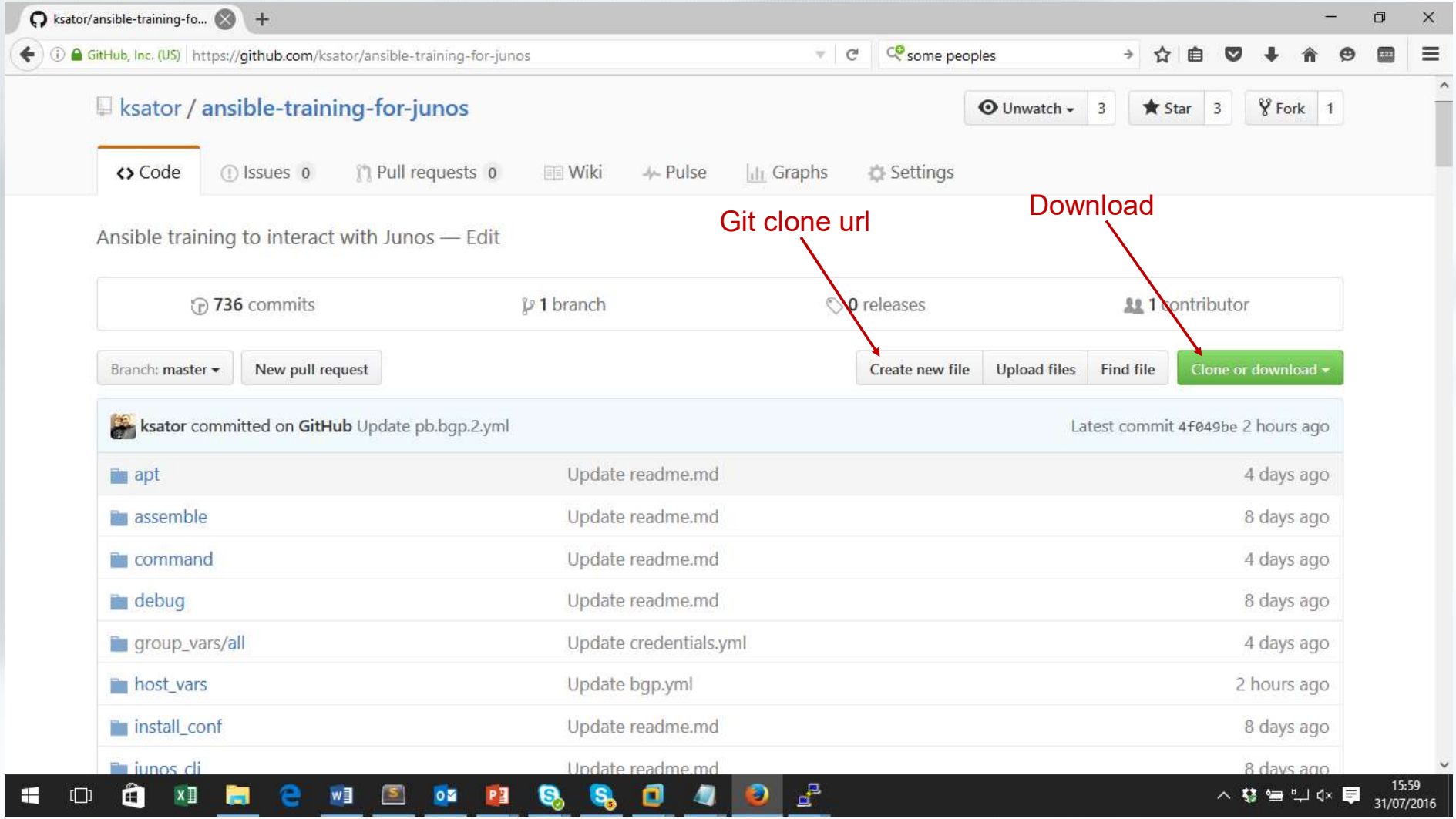
```
---
- name: Create and apply configuration for Leaves QFX / L2
  hosts: leaf-qfx-l2
  connection: local
  gather_facts: no
  roles:
    - common
    - underlay-ebgp
    - overlay-evpn-qfx-l2
    - overlay-evpn-access
```

- Roles are called inside a playbook
- Very useful to reuse the same  automation content across playbook
  - Tasks
  - Templates
  - variables
- You can use the “ansible-galaxy init” command to create the skeleton's role

# ANSIBLE DEMO

# GIT CLONE

- The project https://github.com/ksator/ansible-training-for-junos has many ready to use Ansible playbooks to interact with Junos devices
- Read the instructions in the readme file:
  - https://github.com/ksator/ansible-training-for-junos/blob/master/README.md
- To get the scripts in your laptop, clone it:
  - git clone https://github.com/ksator/ansible-training-for-junos.git
- Playbooks:
  - All playbooks are named pb.*.yml
  - You will find them in different directories. Each directory has a readme file as well.
  - use the ansible-playbook command to execute them



# OTHER REPOSITORIES

- Some others nice repositories regarding Ansible and Juniper:
  - https://github.com/JNPRAutomate/ansible-junos-examples
  - https://github.com/dgjnpr/ansible-template-for-junos
  - https://github.com/JNPRAutomate/ansible-junos-evpn-vxlan
  - https://github.com/JNPRAutomate/ansible-demo-ip-fabric

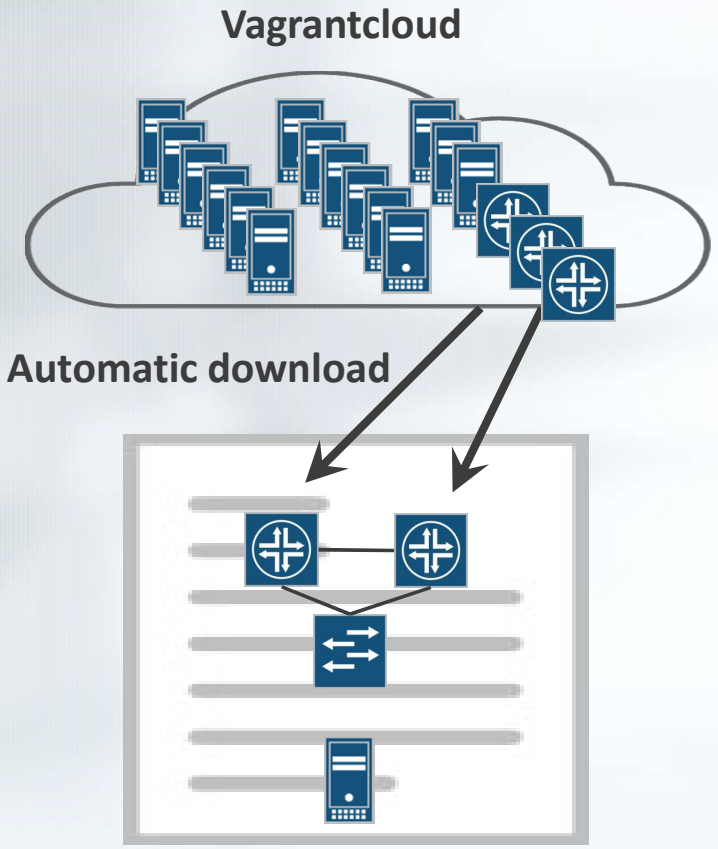
# What is Vagrant ?

- Command line utility for managing the lifecycle of virtual machines
- A tool for building a virtualized environment
  - Tool of choice for DevOps
- From hashicorp:
  - Software company based in San Francisco.
  - HashiCorp provides open source tools and commercial products for datacenter management. vagrant, vault, ....
  - founded by Mitchell Hashimoto
- https://www.vagrantup.com/
- requirements: install vagrant and  install vbox (Virtualbox)

# What is Vagrant ?

A tool for building a virtualized environment

integrated with
Virtualbox/Vmware/docker/hyper-v/…
Ansible/chef/puppet/…
A “Box store” in the cloud




**Vagrantfile**
Define what type of VM/Box
Define the physical topology

# Vagrant providers

- providers:
  - Vagrant uses providers to spin up a virtual machine
  - https://www.vagrantup.com/docs/providers/
  - Was originally tied to VirtualBox (more options today)
  - Vagrant support other hypervisors (but not free and the regular one is Virtualbox)
  - Virtual Machines are provisioned on top of VirtualBox, VMware, ...
  - vagrant is free with Virtualbox (https://www.virtualbox.org/)

# Vagrant provisioners

- provisioners:
  - https://www.vagrantup.com/docs/provisioning/
  - Provisioning tools such as Ansible/Puppet/Chef … can be used to automatically install and configure software on the vagrant boxes

# Provider and provisioning

**Provisioning**




**Providers**


# Vagrant for network engineers

- Vagrant plugins to support of junos boxes:
  https://github.com/JNPRAutomate/vagrant-junos
- public repo with vagrant boxes:
  - https://atlas.hashicorp.com/boxes/search?
  - https://atlas.hashicorp.com/juniper
- Use cases:
  - https://ittechnologist.wordpress.com/2015/09/09/use-vagrant-with-juniper-junos-vms-on-windows/
  - https://keepingitclassless.net/2015/03/go-go-gadget-networking-lab/
  - https://www.dravetech.com/blog/2016/01/08/vagrant-for-network-engineers.html

# Vagrant for network engineers: Use cases

- building a virtualized environment
  - Tool of choice for DevOps
- testing/learning
  - cli
  - Protocols
  - Automation
- running demo
- automated testing of network changes
  - CI/CD
  - IaC: Ubiquitous, inexpensive virtualized network devices for Infrastructure as a Code
    - http://www.networkcomputing.com/networking/achieving-infrastructure-code/1330106672

# vagrantfile

- describes the virtual machines you want, what network interfaces they should have ...
- written in Ruby, but no need to be a Rubist since it is mostly simple variable assignment
- see https://github.com/ksator/vagrant-junos for some examples

# How to start a Vagrant Box

| | |
|---|---|
| vagrant init juniper/ffp-12.1X47-D15.4 | Create Vagrantfile |
| vagrant up | Download and start |
| Vagrant ssh | Connect |

# Vagrant commands

- vagrant init (create vagrantfile)
- vagrant init juniper/ffp-12.1X47-D15.4-packetmode
- vagrant up (download and start)
- Vagrant status
- vagrant ssh
- vagrant halt
- vagrant destroy
- ….

# VAGRANT DEMO

# VAGRANT DEMO

- Visit https://github.com/ksator/vagrant-junos
- Clone this repository https://github.com/ksator/vagrant-junos.git
  - There are several vagrantfiles with vsrx boxes, Ubuntu boxes, ansible provisionner, jinja2 templates ….

# Introduction to Git and Github

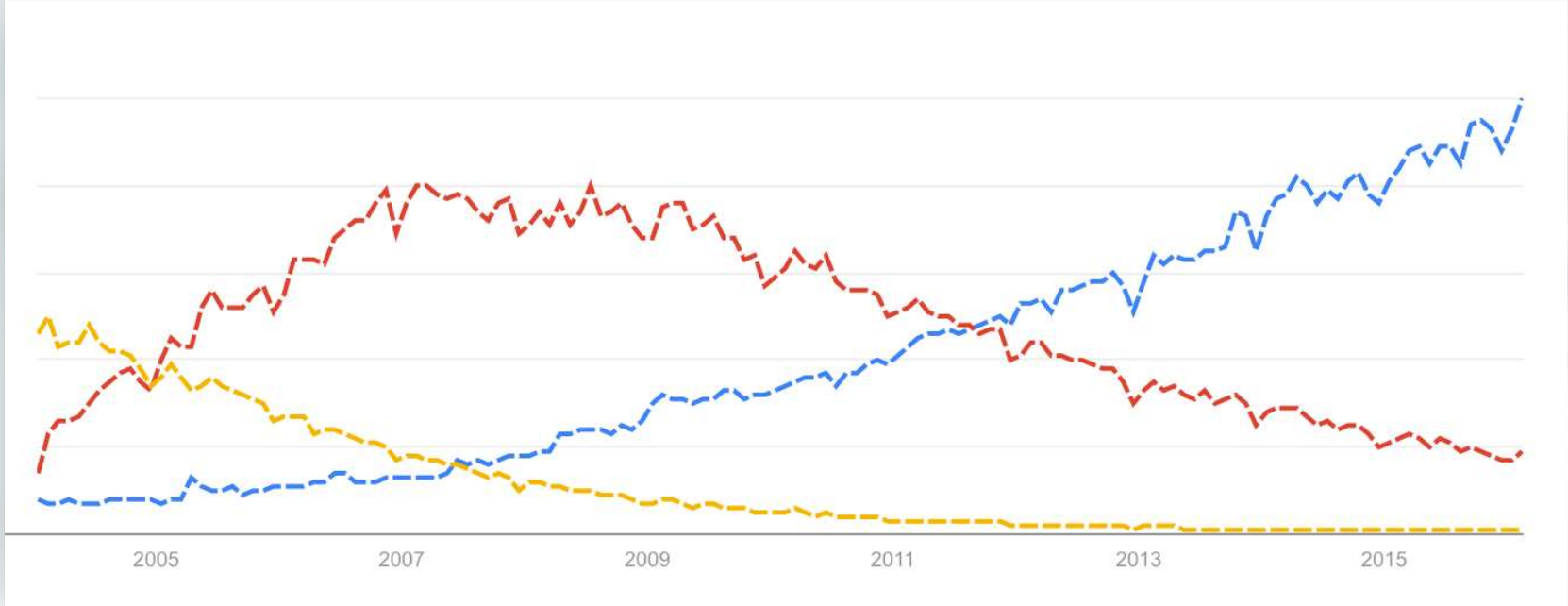
# AGENDA

- **What is Git?**
- **Git Fundamentals**
- **Installing Git**
- **Using Git (with GitHub)**
  **Repo, commits, etc.**

# What is Git?

- version control system
  - a system that tracks changes to files (or groups of files) over time.
  - A repository is a group of files that a version control tracks.
- Distributed/Decentralized
  - multiple systems to host entire copies of the repository
  - allows the users on those systems to collaborate on changes to the repository.
  - creates workflow options
  - Sync, push, pull
  - Read-only, write
- Functions offline
- Workflow does not have a single point of failure due to cloning, which creates a full backup

# Is Git popular ?

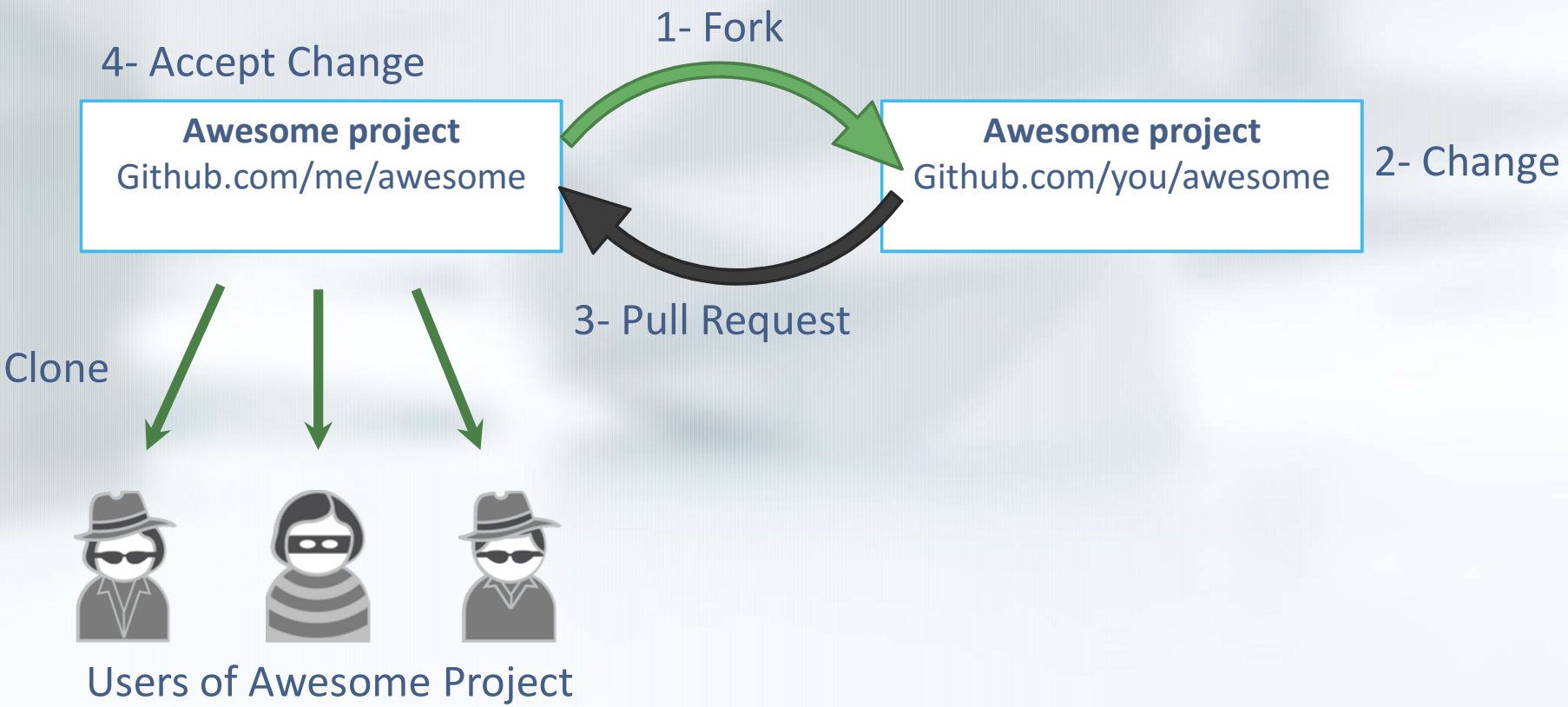



GIT

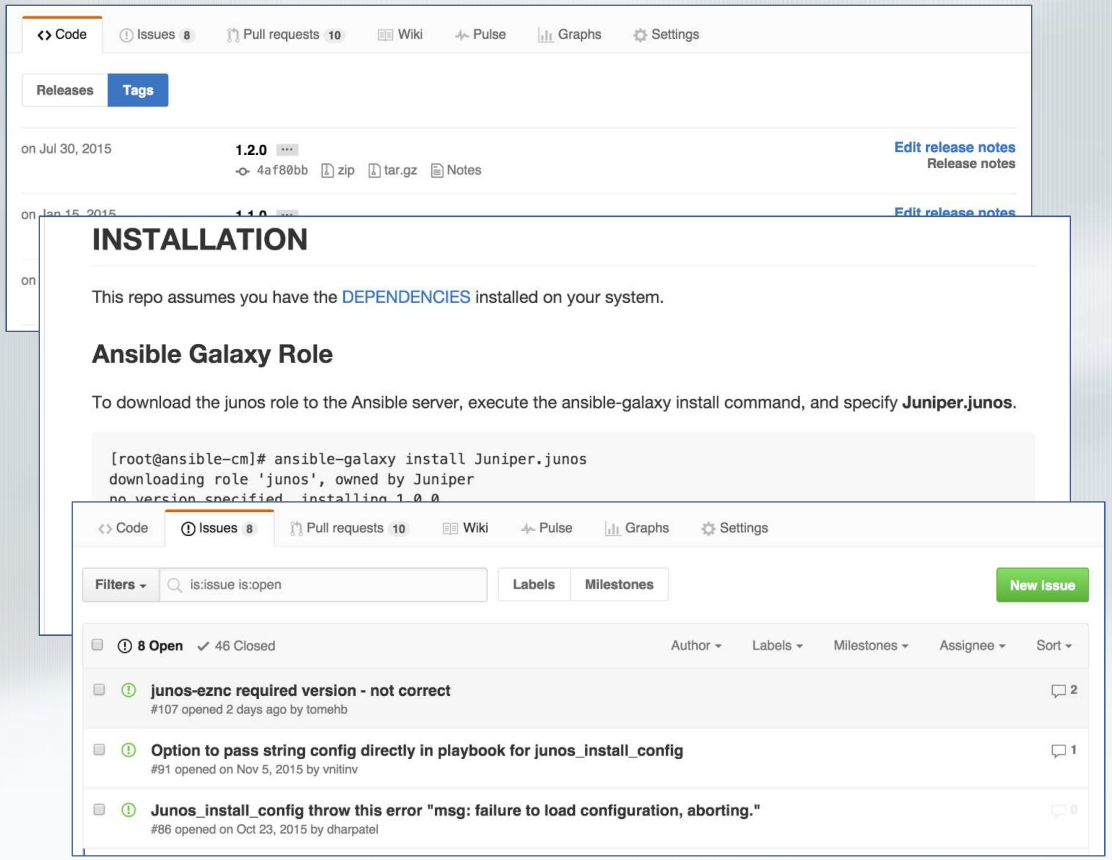
Subversion

CVS

# Design for collaboration (fork)



# More than just Git servers




- Integrate other tools to manage a project
  - Issues management
  - Wiki
  - Integration with third party tools
  - Releases
  - Stats

# CLI / GUI

## CLI

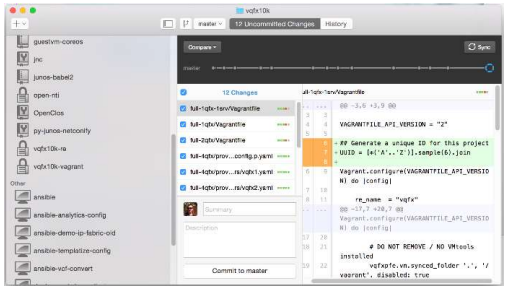
```
[rsherman@localhost example-existing]$ git status
On branch new-feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   functions.py

no changes added to commit (use "git add" and/or "git commit -a")
[rsherman@localhost example-existing]$ git reset --hard
HEAD is now at 16bca75 Updates
[rsherman@localhost example-existing]$ git status
On branch new-feature
nothing to commit, working directory clean
```

- Work with all projects
- Not specific to any solution
- 100% features available

## GUI



- Provided by Github
- Work with all git projects
- Not 100% features available

https://**desktop.github**.com/

# Git Directories: Three states of files

**Committed**

- Data is safely stored in your local database

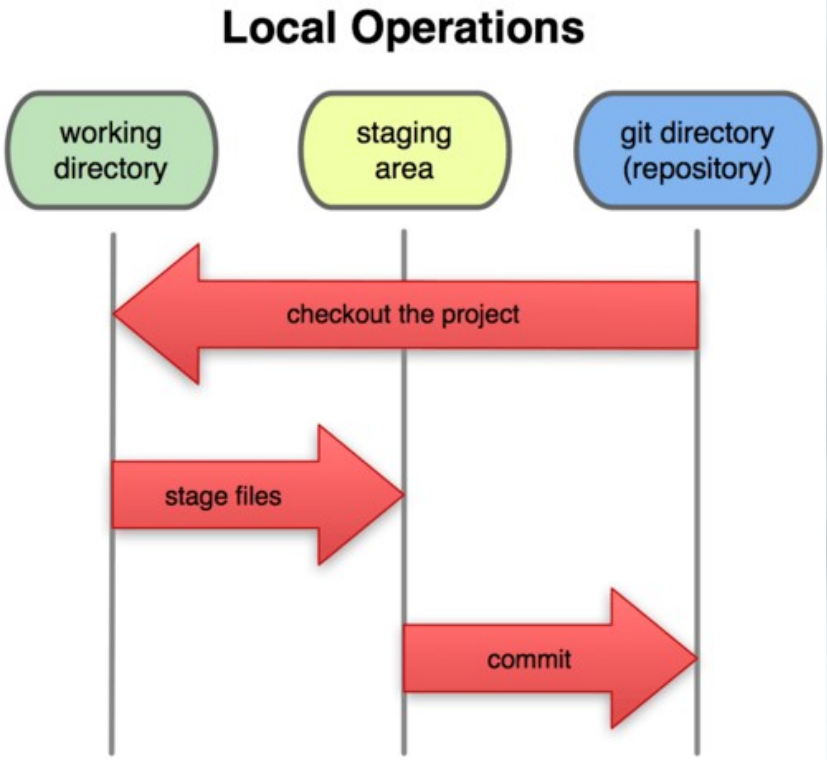
**Modified**

- Files have been changed but not committed to your database

**Staged**

- Modified files have been marked in their current version to go into the next commit snapshot

# Three sections of a Git Project




Three main sections of a Git project:

- The Git directory
- The working directory
- Staging area

# Git Commands (git help)

```
add         Add file contents to the index
bisect      Find by binary search the change that introduced a bug
branch      List, create, or delete branches
checkout    Checkout a branch or paths to the working tree
clone       Clone a repository into a new directory
commit      Record changes to the repository
diff        Show changes between commits, commit and working tree, etc
fetch       Download objects and refs from another repository
grep        Print lines matching a pattern
init        Create an empty git repository or reinitialize an existing one
log         Show commit logs
merge       Join two or more development histories together
mv          Move or rename a file, a directory, or a symlink
pull        Fetch from and merge with another repository or a local branch
push        Update remote refs along with associated objects
rebase      Forward-port local commits to the updated upstream head
reset       Reset current HEAD to the specified state
rm          Remove files from the working tree and from the index
show        Show various types of objects
status      Show the working tree status
tag         Create, list, delete or verify a tag object signed with GPG
```

# Top Five Git Commands (CLI)

```
$ git status
```

Lists all new or modified files to be committed

```
$ git push [alias] [branch]
```

Uploads all local branch commits to GitHub

```
$ git add [file]
```

Snapshots the file in preparation for versioning

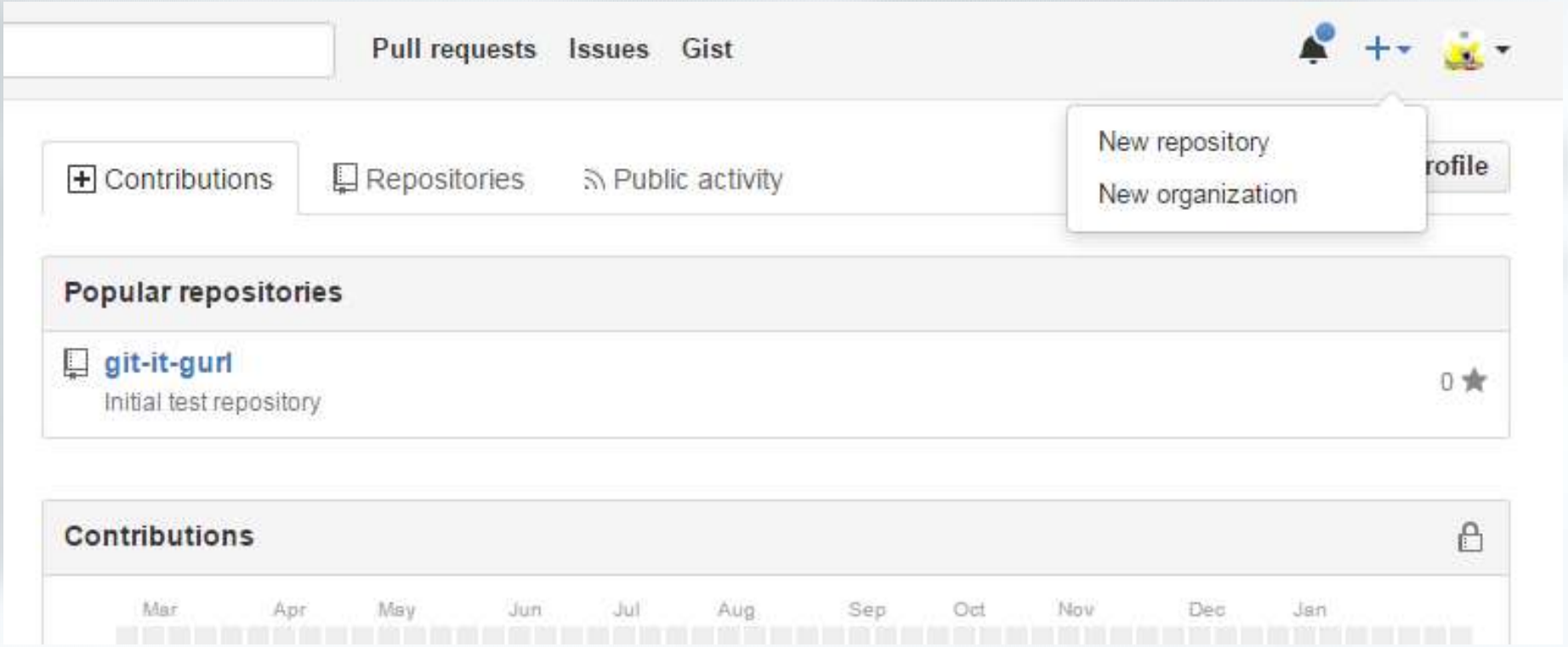
```
$ git pull
```

Downloads bookmark history and incorporates changes

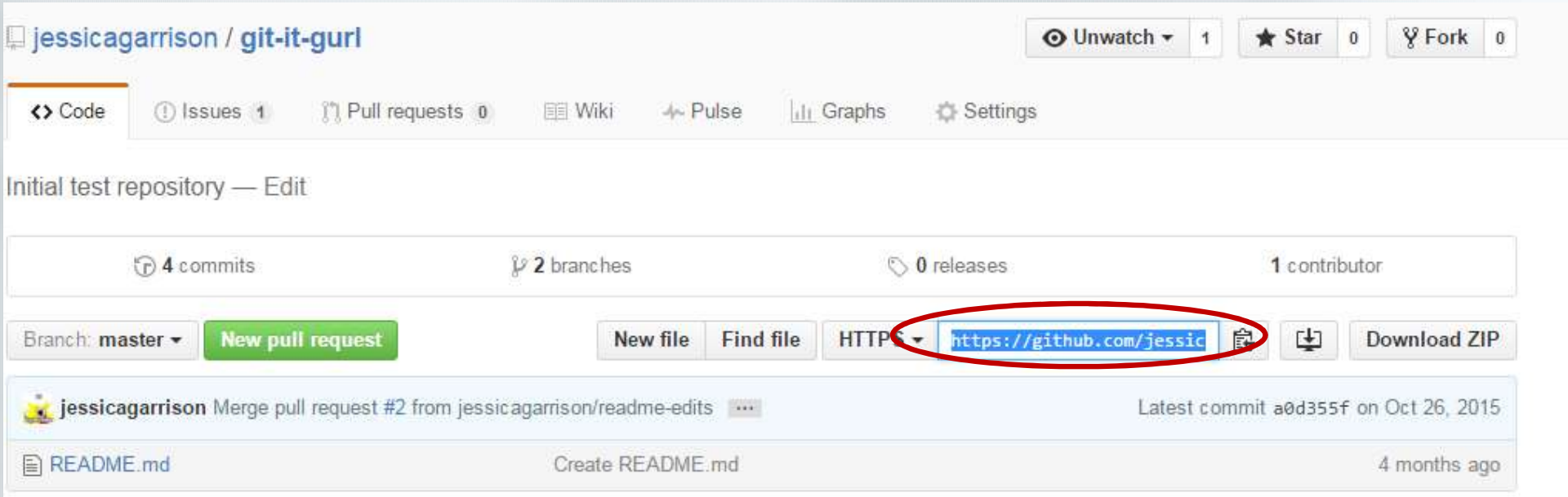
```
$ git commit -m "[descriptive message]"
```

Records file snapshots permanently in version history

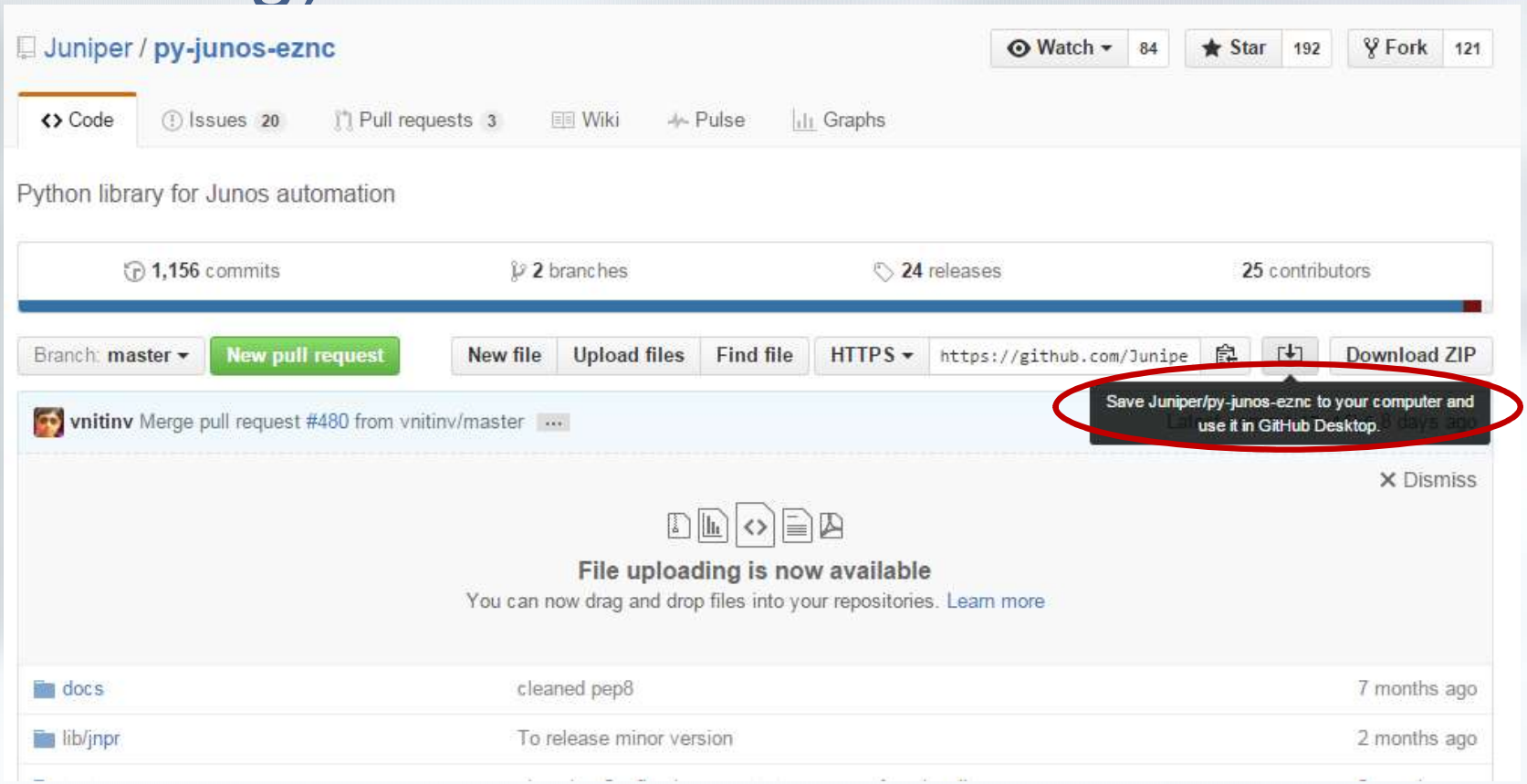
# Creating a Repo



# Get the clone URL



# Save to computer (website variation on cloning)



# Clone a repository (CLI)

```
$ git clone [url]
```

Downloads a project and its entire version history

- One can rename a repository or directory while cloning.
- Sample command:
  - git clone git@github.com:jgarrison/MXtesting.git NewName

# Committing changes (CLI)

```
[rsherman@localhost example-existing]$ vi functions.py
[rsherman@localhost example-existing]$ vi amazing.py
[rsherman@localhost example-existing]$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   amazing.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        functions.py

no changes added to commit (use "git add" and/or "git commit -a")
[rsherman@localhost example-existing]$ git add amazing.py
[rsherman@localhost example-existing]$ git add functions.py
[rsherman@localhost example-existing]$ git commit -m "Updates"
[master 16bca75] Updates
 2 files changed, 1 insertion(+)
 create mode 100644 functions.py
```

# Pushing changes (CLI)

```
[rsherman@localhost example-existing]$ git push origin master
Username for 'https://github.com': shermdog
Password for 'https://shermdog@github.com':
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 309 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/shermdog/example-existing.git
   4788bdb..16bca75  master -> master
```

# Create repositories

## Start a new repository or obtain one from an existing URL

```
$ git init [project-name]
```

Creates a new local repository with the specified name

```
$ git clone [url]
```

Downloads a project and its entire version history

# Make Changes

## Review edits and craft a commit transaction

| |
|---|
| **$ git status** |
| Lists all new or modified files to be committed |
| **$ git diff** |
| Shows file differences not yet staged |
| **$ git add [file]** |
| Snapshots the file in preparation for versioning |
| **$ git commit -m "[descriptive message]"** |
| Records file snapshots permanently in version history |

# Group Changes

## Name a series of commits and combine completed efforts

| |
|---|
| **$ git branch** |
| Lists all local branches in the current repository |
| **$ git branch [branch-name]** |
| Creates a new branch |
| **$ git checkout [branch-name]** |
| Switches to the specified branch and updates the working directory |
| **$ git merge [branch]** |
| Combines the specified branch's history into the current branch |

# Synchronize Changes

## Register a repository bookmark and exchange version history

| |
|---|
| **$ git fetch [bookmark]** |
| Downloads all history from the repository bookmark |
| **$ git merge [bookmark]/[branch]** |
| Combines bookmark's branch into current local branch |
| **$ git push [alias] [branch]** |
| Uploads all local branch commits to GitHub |
| **$ git pull** |
| Downloads bookmark history and incorporates changes |

# CONTINIOUS INTEGRATION

# EXAMPLE WITH PYEZ

- there is  a github webhook with Travis CI to automate the tests.
  - https://github.com/Juniper/py-junos-eznc
- Automated test details with TRAVIS CI:
  - https://github.com/Juniper/py-junos-eznc/blob/master/.travis.yml
- Test results details:
  - https://travis-ci.org/Juniper/py-junos-eznc
  - https://travis-ci.org/Juniper/py-junos-eznc/builds
  - https://travis-ci.org/Juniper/py-junos-eznc/jobs/145493473
- coverage of the automated tests with COVERALLS:
  - https://coveralls.io/github/Juniper/py-junos-eznc

# EXAMPLE WITH JUNIPER ROLES FOR ANSIBLE

- the doc is automatically built based on docstrings (comment/doc in source code) with sphinx and hosted on Readthedocs
- source code https://github.com/Juniper/ansible-junos-stdlib
- source code library https://github.com/Juniper/ansible-junos-stdlib/tree/master/library
- jinja2 used by sphinx to build the doc based on docstrings https://github.com/Juniper/ansible-junos-stdlib/blob/master/docs/rst.j2
- RTD http://junos-ansible-modules.readthedocs.io/en/1.3.1/

# DEMO

- visit https://github.com/ksator/continuous-integration
  - this github has a webhook with Travis CI and coveralls.
  - https://github.com/ksator/continuous-integration/blob/master/.travis.yml
  - read the readme file.
- clone https://github.com/ksator/continuous-integration.git
- TRAVIS CI:
  - https://travis-ci.org/search/continuous-integration
  - https://travis-ci.org/ksator/continuous-integration/builds
  - https://travis-ci.org/ksator/continuous-integration/jobs/143342967
- COVERALLS:
  - https://coveralls.io/github/ksator/continuous-integration
  - https://coveralls.io/builds/6924201/source?filename=maths.py

# ONLINE NETWORK AUTOMATION QUIZ

# ONLINE NETWORK AUTOMATION QUIZ

# ONLINE NETWORK AUTOMATION QUIZ

- You need a connected device (iphone …)
  - Use the QR code
  - Or go to https://kahoot.it/
    - Type the game id
    - Choose a name
- The quiz is public
  - https://getkahoot.com/
  - In order to launch it yourself, you need an account
  - Search for “network automation quiz”
  - There are 2 quiz (short version and long version)


JUNIPer
NETWORKS
THANK YOU!