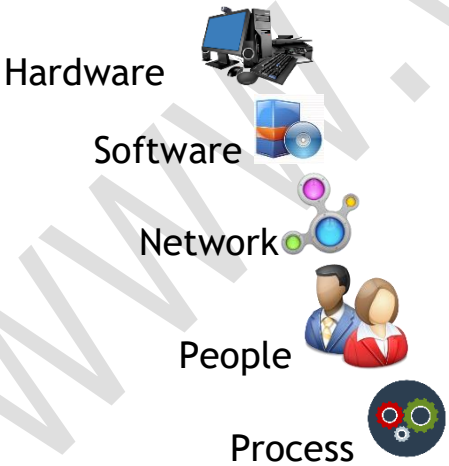
IT infrastructure refers to the composite hardware, software, network resources and services required for the existence, operation and management of an enterprise IT environment. It allows an organization to deliver IT solutions and services to its employees, partners and/or customers and is usually internal to an organization and deployed within owned facilities:



The underlying problem is on how to maintain the **STATE** of the servers in terms of what packages to be installed/removed, which services to started/stopped, creating user accounts, giving permissions, creating dir/files, taking backcup etc.

Configuration Management

Configuration management (CM) refers to the process of systematically handling changes to a system in a way that it maintains integrity over time

What is Ansible?

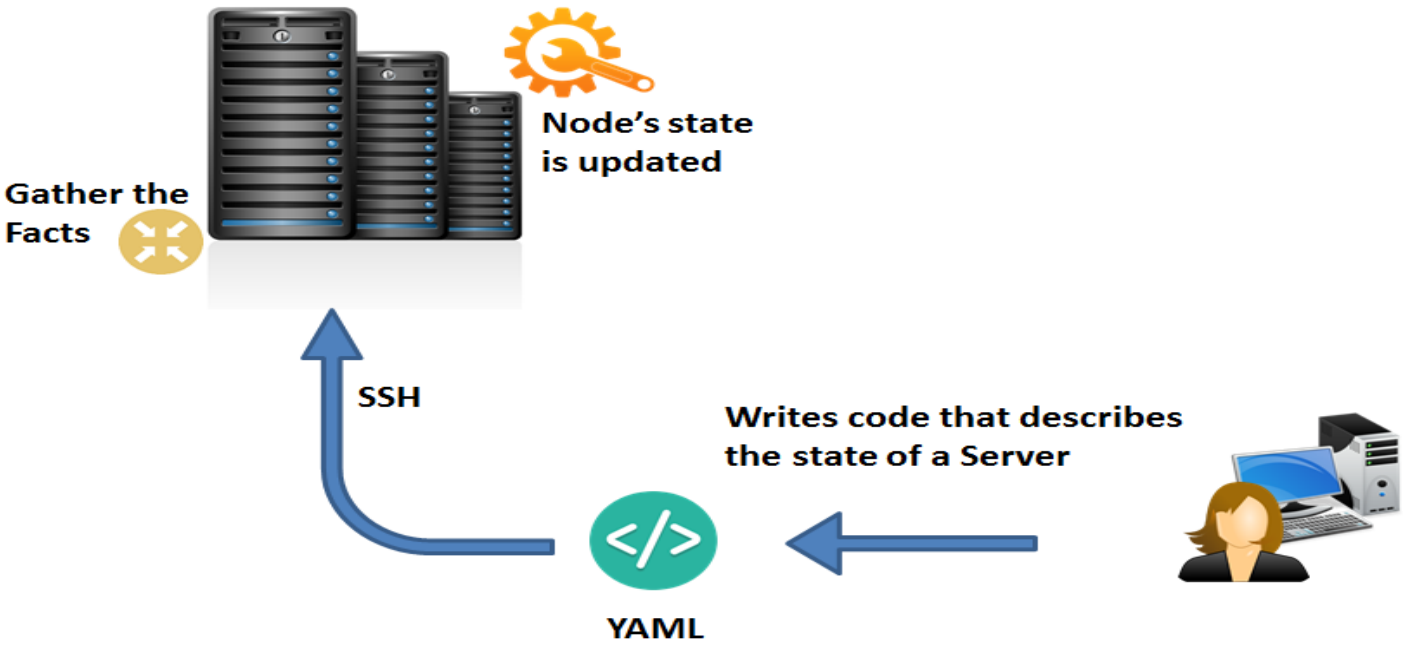
- Ansible is an automation engine that automates software provisioning, configuration management, and application deployment
- Manages infrastructure whether it is on-premises or in the cloud.
- It turns your infrastructure as code i.e your computing environment has some of the same attributes as your application:
  - Your infrastructure is versionable.
  - Your infrastructure is repeatable.
  - Your infrastructure is testable.
- You only need to tell what the desired configuration should be, not how to achieve it

Why Ansible?

- Agentless
- Relies on ssh
- Uses python
- Push mechanism

How Ansible Works?

Ansible works by connecting to your nodes and pushing out small programs, called "Ansible modules" to them. Ansible then executes these modules (over SSH by default), and removes them when finished. Your library of modules can reside on any machine, and there are no servers, daemons, or databases required.



The master node in the above picture is the controlling node (managing node) which controls the entire execution. It's the node from which you are running the installation. The inventory file provides the list of hosts where the Ansible modules needs to be run and the management node does a SSH connection and executes the small modules on the hosts machine and installs the product/software.

Beauty of Ansible is that it removes the modules once those are installed so effectively it connects to host machine, executes the instructions and if it's successfully installed removes the code which was copied on the host machine which was executed

Setup Ansible on AWS(Ubuntu Server)

```
$ apt-get update
$ apt-get install -y ansible
```

Setup Ansible on AWS(Centos Server)

```
$ yum install epel-release
$ yum update
$ yum install git python python-devel python-pip openssl ansible
```

Ansible Environment Setup

- Settings in Ansible are adjustable via a configuration file called “ansible.cfg”. Edit **/etc/ansible/ansible.cfg** & enable the below lines

inventory = /etc/ansible/hosts
sudo_user = root

- Ansible recognizes systems listed in Ansible’s inventory file, which defaults to being saved in the location **/etc/ansible/hosts**
- The format for /etc/ansible/hosts is an INI-like format and looks like this:

[groupname]

machinename|machineIP #Incase of VM#

<MachineName> ansible_host=<<ec2-private-ip>> ansible_user=<<ec2-user>> ansible_ssh_private_key_file=/location/of/the/keypair/your-key.pem #Incase of AWS EC2#

Host Patterns

- Patterns in Ansible are how we decide which hosts to manage or what machines Ansible should connect
- A pattern can usually refer to a particular machine or an groupname
- "all" pattern refers to all the machines in an inventory
- You can refer to hosts within the group by adding a subscript to the group name while giving the pattern

  groupname[0] -- picks the first machine in the group
  groupname[1] -- picks the second machine in the group
  groupname[-1] -- picks the last machine in the group
  groupname[0:1] -- picks first 2 machine in the group

Ansible Ad-Hoc Commands

- Ad hoc commands are commands which can be run individually to perform quick task only once.
- Use **/usr/bin/ansible** to run ad-hoc tasks really quick & don't want to save for later
- These are quick one-liner without writing a playbook

Syntax:

ansible [group|host|all] -m <module_name> -a <arbitrary_cmds>

- List all the hosts under group 'demo'

$ ansible demo --list-hosts

- Ping all the machines under group 'demo'

$ ansible demo -m ping

- List all the files under /home/ansible dir on all the machines under group 'demo'

$ ansible demo -a "ls -al /home/ansible"

- Display the last 10 lines from /var/log/messages file on all the machines under group 'demo'

$ ansible demo -a "cat /var/log/messages"

- Run any tasks with sudo privilege, use -s

$ ansible demo -s -a "cat /var/log/messages"

- Use copy module to copy a file from Ansible master to machines under group 'demo'

$ ansible demo -m copy -a "src=filename dest=filename"

- Install/Remove/Update a Package

$ ansible all -s -m yum -a "pkg=httpd state=present"

$ ansible all -s -m yum -a "pkg=httpd state=absent"

$ ansible all -s -m yum -a "pkg=httpd state=latest"

- Start/Stop/Restart a Service

$ ansible all -s -m service -a "name=httpd state=started"

$ ansible all -s -m service -a "name=httpd state=stopped"

$ ansible all -s -m service -a "name=httpd state=restarted"

- Create/Delete a User account

$ ansible all -s -m user -a "name=adam"

$ ansible all -s -m user -a "name=adam state=absent"

Gathering Facts:

- List all the properties that Ansible gathers while connecting any machine, output will be json format

$ ansible demo -m setup

Ansible YAML Basics

- Ansible uses YAML syntax for expressing Ansible playbooks
- For Ansible, nearly every YAML file starts with a list
- Each item in the list is a list of key/value pairs, commonly called a "hash" or a "dictionary"
- All YAML files can optionally begin with "---" and end with "..."
- All members of a list are lines beginning at the same indentation level starting with a "- "

```
--- # A list of tasty fruits
fruits:
  - Apple
  - Orange
  - Strawberry
  - Mango
```

- A dictionary is represented in a simple key: value form (the colon must be followed by a space)

```
--- # An employee record
Employee:
   name: ADAM
   job: DevOps Engineer
      skill: Elite
```

Ansible Playbooks

- Playbooks are Ansible's configuration, deployment, and orchestration language
- Playbooks describe a policy you want your remote systems to enforce, or a set of steps in a general IT process.
- Playbooks orchestrate steps of any manual ordered process, even as different steps must bounce back and forth between sets of machines in particular orders
- Playbooks are written in YAML format
- **/usr/bin/ansible-playbook** is used for running configurations from an playbook

  Syntax:
  Ansible-playbook <playbook>.yml

Playbooks are divided sections & there are 3 major sections:

1. **Target Section** - Defines the hosts against which playbooks tasks has to be executed
2. **Variable Section** - Defines variables
3. **Tasks Section** - List of all modules that we need to run, in an order

Target Section:

```
---# My first Yaml
- hosts: <host_pattern>
  become: <yes|no>          # default is no #
  become_user: <username>  # user as whom ansible should be executed #
  connection: <ssh|local>  # defaults to ssh #
  gather_facts: <yes|no>   # defaults to yes #
```

Task Section:

```
--- # My First YAML playbook
- hosts: <group>
  become: <yes|no>
  connection: ssh
  gather_facts: no
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
```

Variables Section:

- Refer various items for debug, set constant instead of typing every time
- foo_port is a great variable. foo5 is fine too.
- foo-port, foo port, foo.port and 12 are not valid variable names.
- To use the variable, use the syntax '**{{variablename}}**'

```
--- # My First YAML playbook
- hosts: <group>
  become: <yes|no>
  connection: ssh
  gather_facts: no
  vars:
     <variablename>: <value>
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
```

Handler Section:

- Consists the ability to notify a handler only when state change happens
- Also call another set of tasks

```
--- # My First YAML playbook
- hosts: <group>
  become: <yes|no>
  connection: ssh
  gather_facts: no
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
      notify: <Handler task name>
  handlers:
    - name: <name of the handler task>
      <modulename>: <arbitary commands>
```

Ansible Dryrun

- Check whether the playbook is formatted correctly
- Test how the playbook is going to behave without running the tasks

$ ansible-playbook playbook.yml --check

Example:

```
--- # My First YAML playbook
- hosts: demo
  become: yes
  vars:
    pk: httpd
  tasks:
    - name: Install HTTPD server on centos 7
      yum: name='{{pk}}' state=installed
      notify: Restart HTTPD   # this is called only if the action is ran & successful #
  handlers:
      - name: Restart HTTPD   # this has to match the notify name #
        action: service name='{{pk}}' state=restarted
```

Run ansible-playbook to call the playbook

```
$ ansible-playbook playbook.yml
```

Asynchronous Actions and Polling

- While using Ansible against multiple machines, the operations may run longer than SSH
- While one long task is running, another short task can be executed in asynchronous mode
- Specify the maximum runtime to timeout & how frequently to poll for status
  - ✓ **async**: <seconds to timeout the task>
  - ✓ **poll**: <seconds to poll for the status of the task>

Syntax:

```
--- #Playbook to run task in parallel
- hosts: <group>
  become: <yes|no>
  connection: ssh
  gather_facts: no
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
      async: <seconds>
      poll: <seconds>
```

Example: Check class notes for example

Run Once

- In some cases there may be need to only run a task one time & on one host
- This can achieved by configuring "**run_once**" on a task
- This can be optionally paired with **"delegate_to"** to specify an individual host to execute on

Syntax:

```
--- #Playbook to run task once
- hosts: <group>
  become: <yes|no>
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
      run_once: true
      delegate_to: <node to which the task should run>
```

Example: Check class notes for example

Loops

- Often you'll want to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached
- In those scenarios you will iterate the same task multiple times against different values using "**with_items**" & read the value of each iteration using the ansible variable "item"

Syntax:

```
--- #Playbook to run task in loop
- hosts: <group>
  become: <yes|no>
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
      with_items:
        - Value1
        - Value2
```

Example: Check class notes for example

Conditions

- Few tasks might be needed to execute only on specific scenario
- Sometimes you will want to skip a particular step on a particular host
- In those cases we would use "**when**" statement

Syntax:

```
--- #Playbook to run task based on a condition
- hosts: <group>
  become: <yes|no>
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
      when: <condition to satisfy on when to run the task>
```

Example: Check class notes for example

Capture the task output

- By default Ansible do not capture the output of the tasks
- We have to explicitly store the output into a user variable using the "**register**" statement
- The values will be stored in json format

Syntax:

```
--- #Playbook to run a task and capture its output
- hosts: <group>
```

```
  become: <yes|no>
  tasks:
    - name: <name of the task>
      <modulename>: <arbitary commands>
  register: <variablename>
- debug: var=<variablename>.<attribute>
```

Example: Check class notes for example

Error handling

- By default Ansible stops the execution of the playbook when it finds the first error, so if the first tasks fails to execute then it wont proceed further
- We can skip the failure by using "**ignore_errors**" statement

Syntax:

```
--- #Playbook to ignore errors on task1 and continue running task2
- hosts: <group>
  become: <yes|no>
  tasks:
    - name: <name of the task1>
      <modulename>: <arbitary commands>
      Ignore_errors: yes
    - name: <name of the task2>
      <modulename>: <arbitary commands>
```

Example: Check class notes for example

wait_for - Waits for a condition before continuing

- You can wait for a set amount of time to complete a task
- Waiting for a port to become available is useful for when services are not immediately available

Syntax:

```
--- #Playbook to wait for a port to be available
- hosts: <group>
  become: <yes|no>
  tasks:
    - name: <name of the task1>
      <modulename>: <arbitary commands>
    - name: <name of the task to wait for a port available>
      wait_for:
        port: <port#>
        state: started
    - name: <name of the task to wait for a file to be available>
      wait_for:
        path: <filepath>
```

Example: Check class notes for example

Tags

- If you have a large playbook it may become useful to be able to run a specific part of the configuration without running the whole playbook
- Use the statement “tags” to add a name to a task
- A task can have multiple tag names
- Same tag name can be share with multiple tasks to group them

Syntax:

```
--- #Playbook to wait for a port to be available
- hosts: <group>
  become: <yes|no>
  tasks:
    - name: <name of the task1>
      <modulename>: <arbitary commands>
      tags:
        - <tagname1>
    - name: <name of the task1>
      <modulename>: <arbitary commands>
      tags:
        - <tagname2>
```

- If you want to run a playbook without certain tasks

  $ ansible-playbook playbook.yml --tags "tagname"

- If you want to run a playbook without certain tasks

  $ ansible-playbook playbook.yml -skip-tags "tagname"

Example: Check class notes for example

Ansible Vault

- Ansible allows keeping sensitive data such as passwords or keys in encrypted files, rather than as plaintext in your playbooks
- To Run the playbook which is password protected using Ansible vault, use "--ask-vault-pass" while calling the playbook or "--vault-password-file FILE". Where FILE is the name of file in which password is stored
- Creating a new Encrypted Files

  $ ansible-vault create playbook.yml
- Edit the Encrypted File

  $ ansible-vault edit playbook.yml
- Change the password

  $ ansible-vault rekey playbook.yml
- Uncrypt the file

  $ ansible-vault decrypt playbook.yml
- Encrypt an existing file

  $ ansible-vault encrypt playbook.yml

Ansible Roles

- Adding more & more functionality to the playbooks will make it difficult to maintain in a single file
- We can organize playbooks into a directory structure called roles
- This is already possible by 'include' directives however Roles are automation around it
- Default path for Roles **/home/ansible/playbooks/roles:/etc/ansible/roles:<PWD>**
- We can alternatively keep the master playbook in a different location & specify the Role path in ansible.cfg
- In the /etc/ansible/ansible.cfg, uncomment roles_path & add the roles dir

  roles_path = /home/ansible/playbooks/roles

Framework/Syntax:

masterplaybook.yml

roles/<rolename>/

tasks/main.yml

vars/main.yml

handlers/main.yml

default/main.yml

meta/main.yml

Syntax:

```
--- #Playbook for calling a role
  - hosts: <group>
    become: <yes|no>
    roles:
        - <Rolename>
```

Example: Check class notes for example

Connecting to AWS using Ansible

- Connecting to AWS programmatically requires specific AWS Access Key Id & Secret Access Key

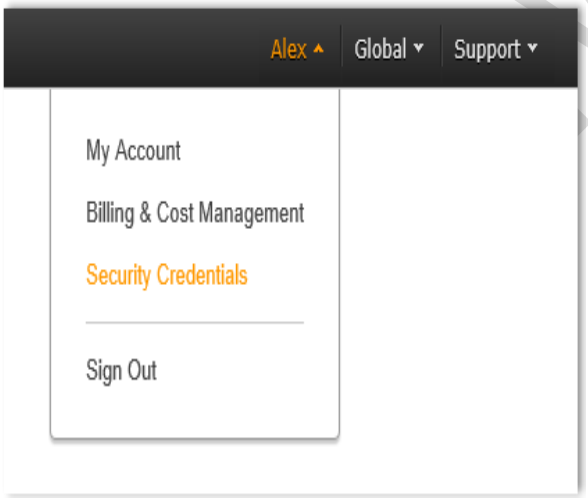
- Create ~/.boto & put the values obtained from the below steps:

[Credentials]
aws_access_key_id = <your_access_key_here>
aws_secret_access_key = <your_secret_key_here>

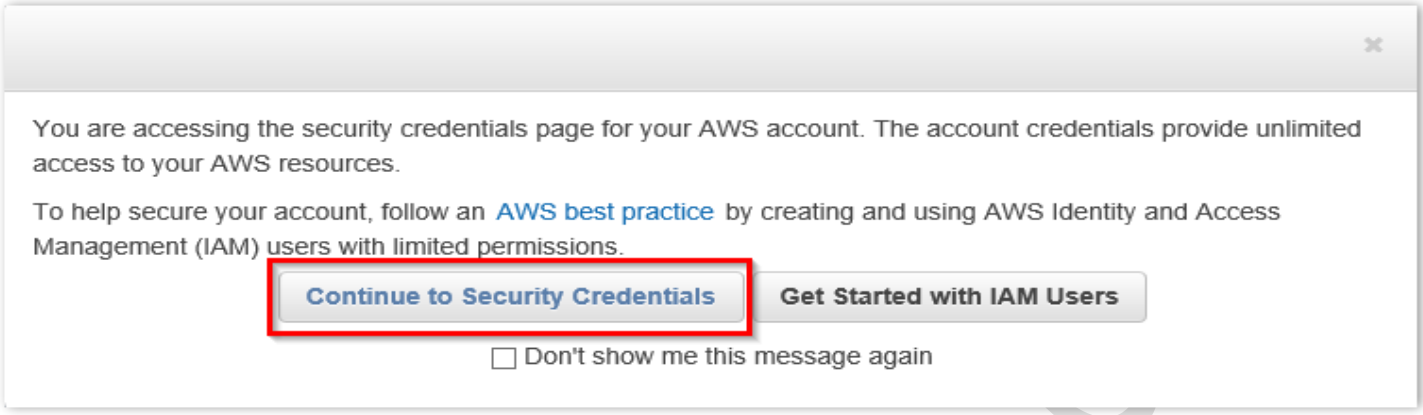
- Copy the keypair.pem file to the Ansible Master server

How to find your AWS Access Key ID and Secret Access Key

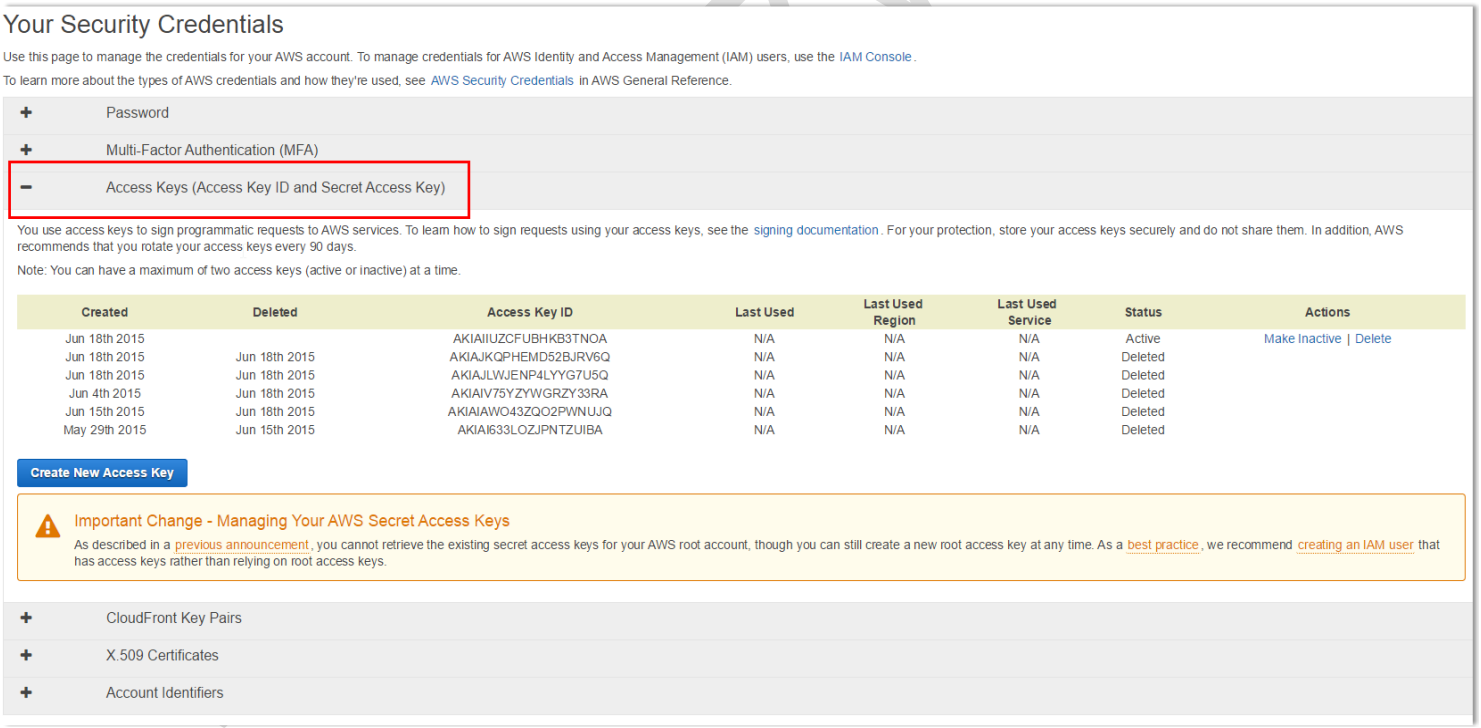
- ✓ Go to Amazon Web Services console and click on the name of your account (it is located in the top right corner of the console). Then, in the expanded drop-down list, select Security Credentials.



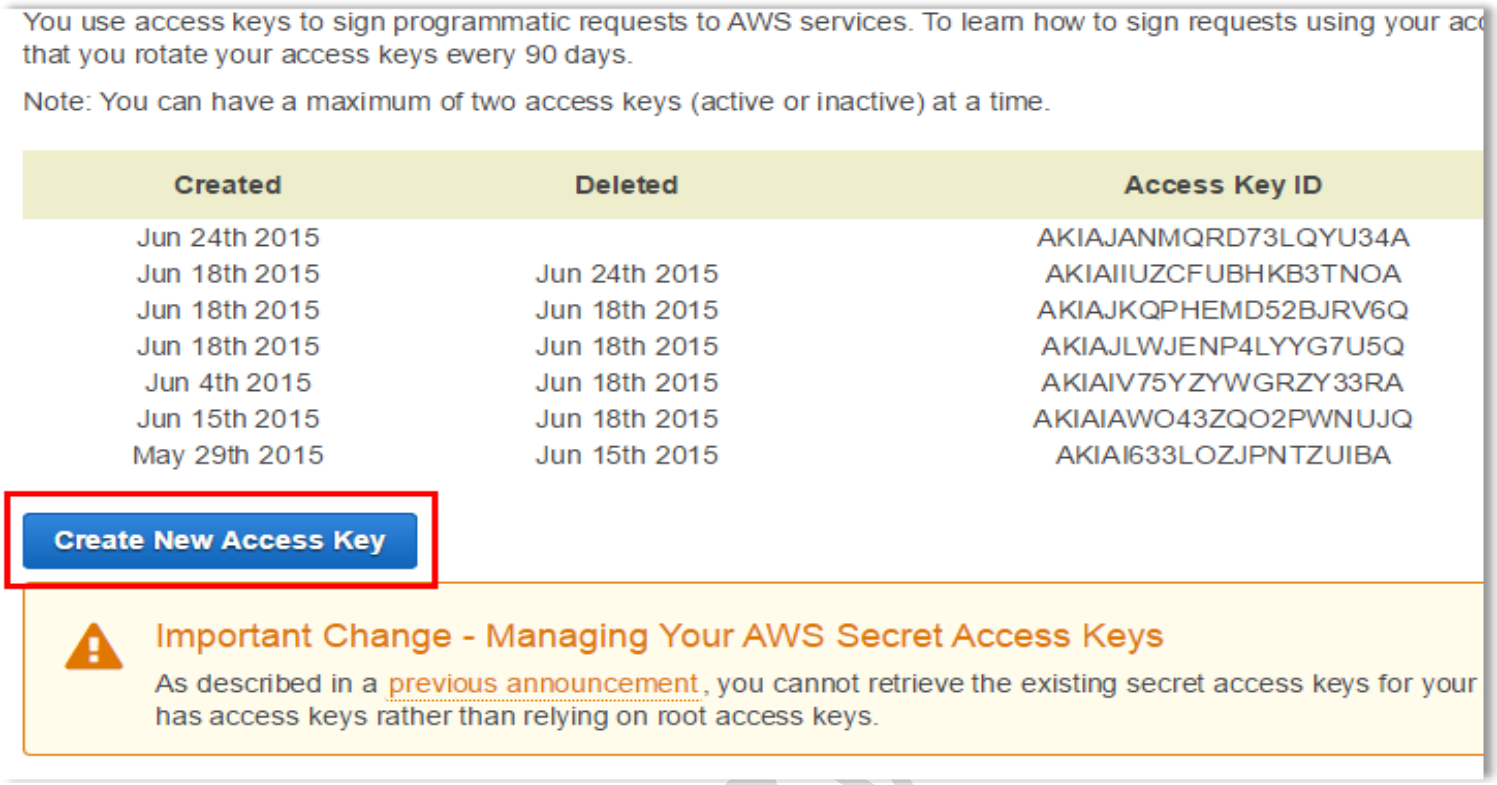
✓ Click the Continue to Security Credentials button



✓ Expand the Access Keys (Access Key ID and Secret Access Key) option. You will see the list of your active and deleted access keys



✓ To generate new access keys, click the Create New Access Key button.



- ✓ Click Show Access Key to have it displayed on the screen. Note, that you can download it to your machine as a file and open it whenever needed. To download it, just click the Download Key File button.

Creating EC2 Instances in AWS using Ansible

```
--- # Creating EC2 Instances in AWS
- hosts: demo
  become: yes
  connection: local
  tasks:
  - name: Install python-pip library  # prerequisite
    apt: name='{{item}}'
    with_items:
     - python-pip
     - python-dev
  - name: Install python-boto library
    pip: name=boto
  - name: Create AWS Instances
    ec2:
     key_name: "wezva"
     instance_type: "t2.micro"
     image: "ami-c58c1dd3"
     wait: true
     region: "us-east-1"
```

Creating/Deleting S3 Buckets in AWS using Ansible

```
--- # Creating EC2 Instances in AWS
- hosts: demo
  become: yes
  connection: local
  tasks:
  - name: Install python-pip library  # prerequisite
    apt: name='{{item}}'
    with_items:
     - python-pip
     - python-dev
  - name: Install python-boto library
    pip: name=boto
  - name: Create S3 Bucket
    S3_bucket:
     name: mys3bucketfortest
     region: “us-east-1”
     state: present   #use state as absent to delete the S3 bucket #
```