Chapter No. 4
"Error Handling, Rollback, and Reporting"
In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.4 "Error Handling, Rollback, and Reporting"
A synopsis of the book’s content
Information on where to buy this book

About the Author’s

Madhurranjan Mohaan is a passionate engineer who loves solving problems. He has more than 8 years of experience in the software industry. He worked as a network consultant with Cisco before starting his DevOps journey in 2011 at ThoughtWorks, where he learned the nuances of DevOps and worked on all aspects from Continuous Integration to Continuous Delivery. He is currently working at Apigee, where he enjoys dealing with systems at scale.

Madhurranjan has also worked with various tools in configuration management, CI, deployment, and monitoring and logging space, apart from cloud platforms such as AWS. He has been a presenter at events such as DevOpsDays and Rootconf, and loves teaching and singing. He is a first-time author and wishes to share his learning with readers from across the world.

I'd like to thank the following folks (in alphabetical order) who've been instrumental in making me what I am in this domain today: Ajey Gore, Chirantan Mitra, Ranjib Dey, Rohith Rajagopal, Sharath Battaie, and Sriram Narayanan. Special thanks to Shreya who was very excited and said, "My kaka (uncle), the author!"

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Ramesh Raithatha is a DevOps engineer by profession and a cyclist at heart. He is currently working at Apigee and has worked on various facets of the IT industry, such as configuration management, continuous deployment, automation, and monitoring. He likes exploring stunning natural surroundings on two wheels in his leisure time.

I would like to thank my friends, without whom we would have completed the book much earlier.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Learning Ansible

Ansible is one of the most popular tools today in the Infrastructure Automation space. It is an IT orchestration engine that can be used in several subverticals, such as configuration management, orchestration, provisioning, and deployment. When compared to other automation tools, Ansible brings you an easy way to configure your infrastructure without the overhead of a client setup.

We started using Ansible to set up our build agents for Continuous Integration (CI) systems and were soon able to set up close to 150 agents that had different applications, within a couple of weeks. At that stage, we primarily used it for configuration management of our build agents, but once we had tasted success, we realized that it would help us solve our problems in production as well. In production, we used another tool for configuration management for a long time, but we needed something else that would help us in orchestration and provisioning. Our deployments and orchestrations were very manual in nature and quite error-prone. We adopted Ansible in our production environments to solve this very problem at the start of the year, and we've had great success so far. We're able to build and rebuild infrastructures at a much faster rate that's less error-prone, involves less manual effort, and uses just a single click. Our deployment times have drastically come down. We've not yet attained Nirvana in this area but we're in a way better situation today compared to a year ago. We learned a lot in the process and we'd like to share our experiences with Ansible in this book. The Ansible website has been of great help along with the very intelligent community of Ansible.

We'd like to talk briefly about DevOps before moving ahead with Ansible. DevOps has been a movement aimed at bringing development and operations teams in organizations together a lot more than usual to release software earlier. Before the DevOps movement came into being, for a long time the notion was that development teams implement the features and throw the software over the wall for the operations team to then take it and run it in production. As a result, the operations teams had no idea what they were deploying, and the developers didn't know how their software was going to be deployed!

Another common complaint that most operations folks have heard is a developer coming to them and saying, "It works on my machine". An operations person recently responded to a similar question by saying, "Let's take your laptop and put it in production". Jokes apart, this isn't the ideal situation, is it? There has been a constant effort to get the development teams and operation teams to work a lot closer through a combination of tools and culture. Without the right culture, irrespective of what tools you use in your organization, the impact isn't going to be massive. However, changing and adopting the best practices between teams and breaking the wall, so to speak, between these diverse teams is what the DevOps movement advocates.

For More Information:

www.packtpub.com/networking-and-servers/learning-ansible
Several teams have begun to embed operations folks in their development workflows, right from the planning phase, so that the operations teams are aware of what's going on; at the same time, they can offer their expertise during the early stages of development. This is an aspect of culture that you can bring about in your organization and that would reduce the risk and help you release your software faster to production. For example, insights around scaling, database replication, and logging practices are skills that operations teams bring in. On the other hand, skills such as testing individual components and using unit, integration, and functional testing are what the operations teams can ideally pick up from the developers.

The other aspect of DevOps that we mentioned earlier was the tools. With the advent and adoption of cloud, which basically introduced the concept of "on-demand" and "pay-as-you-go" computing, tooling has become all the more important. There are primary focus areas, however, where tools have made significant progress. Let's broadly look at these areas:

- **Configuration Management**: Several tools have come up in this area. The aim of configuration management is to make sure your machines attain the intended state as described in the configuration in the fastest possible time and in the right manner so that they can play a particular role in your environment. For example, if you have to introduce a new web server during a traffic surge, how quickly can you do it, once you have a machine is what configuration management addresses. This also has resulted in the operations folks writing code and it's commonly termed as "Infrastructure as code", since all the code that is necessary to set up your infrastructure is now stored in the source control. This has slowly led to the adoption of Software Development Lifecycle (SDLC) for infrastructure code. This includes tools that aid your infrastructure testing. Tools in this space include CFEngine, Chef, Puppet, Ansible, Salt, and so on. Infrastructure-testing tools include Serverspec, Test kitchen, and so on.

- **Provisioning**: The tools in this space address how quickly you can bring up new machines in your data center, virtualized environment, or your cloud. Almost all cloud providers have APIs. The tools in this space use these APIs to speed up instance provisioning. For organizations that are on Linux, containers have made rapid strides in the last year or so with solutions such as Docker and LXC in the forefront, and more and more people are beginning to use tools to make sure their containers are provisioned in an automated way. Ansible plays an important role in both these scenarios. Finally, there are tools such as Vagrant, which help you automate development and test environments.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
• **Deployment**: Tools in this area focus on how you can deploy applications with zero downtime and in the best possible way. Many organizations now perform Rolling deployments or Canary deployments. Ansible supports both. Deployment pipelines have also become popular and tools such as ThoughtWorks Go, Atlassian Bamboo, and Jenkins with its innumerable plugins are strong players in this area.

• **Orchestration**: Tools in this area focus on how to coordinate among various components in your infrastructure so as to perform deployments. For example, making sure a web server is disabled from a load balancer, before releasing a new version of your software to the web server, is a common and famous example. Ansible, Mcollective, Salt, Serf, and Chef are some of the tools that help in this area.

• **Monitoring and Alerting**: Monitoring and alerting tools have evolved to handle fast-growing massive server environments. Legacy tools such as Nagios, Ganglia, and Zenoss along with newer tools such as Graphite, Sensu, and Riemann play a major role in this domain.

• **Logging**: Centralized logging makes sure you collect the right set of logs across systems and applications so that you can write rules on top of them to perform intelligent deductions, be it root cause analysis or alerting. Tools such as Logstash-Kibana, SumoLogic, and Rsyslog are quite popular in this space.

Ansible plays a significant role in four of the six major areas where tooling plays a very important role. Along with this, the people who can contribute heavily to these areas include sysadmins, operations teams, infrastructure admins, developers, and anyone with the mindset of bringing about infrastructure automation. The book aims to help and guide all these categories of users to set up robust automation for their infrastructure.

**What This Book Covers**

*Chapter 1, Getting Started with Ansible*, teaches you the basics of Ansible, its architecture, and how to set it up and get started. It starts with Ansible's "Hello, world!" program and builds the rest of the examples on top of it. You'll be introduced to inventories, modules, variables, and playbooks, and how Ansible can be used for configuration management.

*Chapter 2, Developing, Testing, and Releasing Playbooks*, will focus on how you can develop your Ansible playbooks, test them, how to handle multiple environments, and how to release your Ansible code into production. It also discusses the Software Development Life Cycle (SDLC), which is as important with an infrastructure management tool development as it is with any other custom software that is built.
Chapter 3, Taking Ansible to Production, focuses on all the important features that you
would require for taking Ansible into production, more from a configuration management
perspective. You will learn about features such as include, loops, and conditions in
Ansible; handlers and security management with Ansible; and, most importantly, how
to model your infrastructure using Roles. With the knowledge gained from the first three
chapters, you will know enough to write playbooks that can be deployed in Production
to configure your infrastructure.

Chapter 4, Error Handling, Rollback, and Reporting, helps you with topics such as how
to debug, rollback, and report what you have in production. In almost all cases, you need
to have enough information regarding these topics. It introduces Callback plugins and
techniques you can use to rollback configurations when something goes wrong. It shows
you how you can integrate Ansible with alerting and monitoring tools, and generate
metrics for error handling and reporting. In short, you will be introduced to all that you
would like in your sysadmin avatar.

Chapter 5, Working with Custom Modules, runs you through how you can write custom
modules for your setups. One of the strengths of Ansible is that you can write custom
modules in any language that has an interpreter (if the language is available on the box),
provided that they return a JSON output. In most cases, we have seen that, having
intelligent modules reduce the size of your playbooks and make them more readable.

Chapter 6, Provisioning, explains how you can bring up new instances in clouds as
part of your provisioning activity. With the advent of cloud, the demand for spawning
machines in clouds, such as AWS, Rackspace, and DigitalOcean, have gone up quite
significantly. We'll also look at one of the most exciting container technologies, Docker,
and how you can use Ansible to provision new Docker containers.

Chapter 7, Deployment and Orchestration, looks at how you can deploy Rails as well
as Tomcat applications, along with packaging and deployment techniques. For a large
infrastructure, it's important to deploy software in as little time as possible and, in almost
all cases, with zero downtime; Ansible plays a key role in deployment and orchestration.
We'll also look at how you can use the Ansible pull when you have large infrastructures.

Appendix, Ansible on Windows, Ansible Galaxy, and Ansible Tower, discusses Ansible's
support for Windows. In addition, we'll cover Ansible Galaxy, a free site from where you
can download community roles and get started, and finally, Ansible Tower, a web-based
GUI developed by Ansible that provides you with a dashboard to manage your nodes
via Ansible.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
“If you shut the door to all errors, truth will be shut out.”

-Rabindranath Tagore

The quote here applies to everything in life and not just infrastructure! However, since we're going to continue talking about infrastructure with a key focus on error handling, rollback, and alerting in this chapter, we thought we'd start the chapter on a philosophical note.

So far, we've seen how to write production-ready Ansible playbooks from a configuration management perspective and test the same to overcome any failure. Any time you write code, it's important to think of error handling, logging, alerting, and reporting and if it's based on infrastructure, it becomes even more important. Error handling helps you control the flow of your automation and allows you to notify your users if something goes wrong. Rollbacks become important when the intended task fails and you want to restore the state of the automation to a well-known stable state. Reporting and alerting go hand in hand and we'll talk about it more when we reach that section. We'll also introduce a powerful concept called Callbacks and see how helpful they can be when we run our infrastructure with Ansible.

In this chapter, we will cover the following topics:

- Error handling and Rollback
- Callback plugins: a reporting example
- Monitoring and alerting

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Error handling and Rollback

Automation always comes with a wide range of advantages such as increased productivity, better control of systems, and solid workflows. However, with advantages, there are always disadvantages! You need to have good control over errors that might occur during automation. While you're writing the functional code, to overcome such situations you would generally use a try-catch technique, where you try to run your code, catch errors when your code fails, try to handle them gracefully, and if possible, continue with the flow. Ansible doesn't provide a try-catch feature; rather, it leaves it up to the user to handle errors. For certain types of tasks, you can write custom modules that can leverage the power of your programming language's error handling (We will look at custom modules in the next chapter). However, for other Ansible core modules, you can use Ansible's conditional statements or ignore errors (as was shown in the earlier chapters).

Suppose you have a playbook with many tasks. There is a possibility that a certain task might fail. If this is one of the initial tasks where you validate certain prerequisites, then error handling might not make sense. For example, let's say you're dealing with one of the cloud systems (for example, Amazon Web Services (AWS)) and you have a policy within your organization that whoever utilizes any Ansible playbook to perform cloud-related activities should expose their access keys via environment variables. Hence, as part of your Ansible role or playbook, you might have a common or prerequisite playbook or role that will check whether your access keys are available as environment variables.

Another prerequisite check that we've often seen is users checking if they have the required sudo credentials on machines where the playbooks will run (We've even seen an example related to the sudo check in Chapter 3, Taking Ansible to Production.). This might be required for configuration management tasks and your prerequisite playbook or role would check whether sudo access is first enabled. If your tasks fail at this level, then error handling does not make sense!

However, if your playbook fails after certain tasks have been executed, and they've changed the state of the remote system (remember ok and changed?), then it might cause an outage or unpredictable behavior, as the remote system is not in the intended state. This is definitely a cause for worry if you're one of the key people managing the infrastructure. You need not worry if all tasks were in the ok state, as it only means that they were idempotent, but if a task is running for the first time, yes, you need to worry! A classic example might be when certain services don't come up as expected and the machine is already in the load balancer and you end up dropping live traffic.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
To handle such errors, we recommend writing a rollback playbook. A rollback playbook will include all the tasks that need to be run if the original playbook fails and restore the playbook to the state from where you can serve traffic. The idea here is to run the primary playbook, and if it fails, run the rollback playbook, which will restore the state on the remote host, resulting in a working system at the end.

Let's consider two practical examples that we've seen from close quarters. Suppose you want to update the memcached package on your system, which is being monitored by monit (we've already discussed monit in Chapter 3, Taking Ansible to Production). Now, in order to update memcached, firstly, you will need to stop the monit service so that it doesn't try to start the memcached service, and then you will have to update your memcached package.

Let's say that when you ran your primary playbook, it successfully stopped monit and memcached, but failed while it tried to update the memcached package. Since the memcached and monit services are stopped, it will affect what is cached and might cause monitoring alerts due to increased response times on the server. The memcached package is the frontend. How do you fix this? Add a rollback playbook. The rollback playbook consists of two tasks: start memcached after ensuring the older package is still present on the system, and then start the monit service.

Another such example would be in a hosted scenario. Let's say, you have a customer X, who is configured to run behind a load balancer with two equally balanced servers. Now, if one of your engineers initiates a restart on the first server, then the CPU load will spike on the other server. To take care of the spike, let's say another engineer initiates a restart on the second server while the first one is still not serving the traffic. Now, both your servers are down and your entire traffic towards that load balancer will be affected, resulting in a downtime for your customer.

In order to deal with such human errors, you can use locking. Whenever a restart is initiated on one of the servers, make sure that you lock the load balancer, which is the shared resource, using a tool such as etcd, and unlock it only after the restart is completed and the server can serve the traffic. In the meantime, no one can restart the second server until the first server releases its lock. There is also an aspect of making sure the traffic isn't unmanageable on the second server in such situations when there are other complexities around it. However, basically, you want to make sure that a shared resource is locked while performing such operations.

Note that etcd is a highly available key-value store similar to Apache ZooKeeper.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Let’s see how we can use locking with the Apache Solr restart for multiple servers that are part of the same load balancer, as shown in the following screenshot:

```
$ cat playbooks/solr_restart.yml

```

```
- name: Trying to obtain lock with the following lockname and lockvalue.
  debug:
    msg: "lock name={{ load_balancer }} lock value={{ inventory_hostname }}"

- name: Locking {{ load_balancer }}
  local_action:
    uri:
      url: http://localhost:8983/solr/key/16/{{ load_balancer }}
      method: PUT
      data: "lock name={{ inventory_hostname }}"
      status_code: 201
      headers: HEADER: Content-Type: application/x-www-form-urlencoded
      return_content: yes

- name: Restarting solr
  service:
    name: solr
    state: restarted
    sudo: yes

- name: Unlocking {{ load_balancer }}
  local_action:
    uri:
      url: http://localhost:8983/solr/key/16/{{ load_balancer }}
      method: DELETE
```

In the preceding playbook, we first print the lock name and the lock value for readability. We then go ahead and try to lock the load balancer so that no one else can restart any other server behind that load balancer. If the locking task fails, then it means the load balancer is already locked and someone else has restarted one of the Apache Solr servers behind it. However, if we are successful in obtaining the lock, we restart the solr service and then unlock the load balancer. Apart from these steps, you can also add some checks to this playbook before unlocking the load balancer to make sure that the service is actually up and ready to serve the production traffic.

### Executing the playbook

Consider the following screenshot and let’s see how the `solr_restart.yml` playbook works:

---

For More Information:

www.packtpub.com/networking-and-servers/learning-ansible
As expected, the preceding playbook:

1. Locked the load balancer.
2. Restarted the **solr** service.
3. Unlocked the load balancer.

This was an example of a successful Ansible run. However, what if Ansible was able to acquire the lock but could not start the service, and the playbook run fails as a result? Let's see what happens when the **solr** restart fails. This is demonstrated in the following screenshot:

For More Information:

www.packtpub.com/networking-and-servers/learning-ansible
As you can see, Ansible acquired the lock but could not restart the service. Now, the solr service is not running and the load balancer is locked as well, which will not allow anyone else to restart the other servers from the same load balancer. This is where rollback comes into the picture. In order to roll back, you first need to try to restart the service, and then unlock the load balancer. Let's see an example playbook for rollback in the following screenshot:

In the preceding playbook, we first check if the lock still exists. We have nothing to do if there is no lock, in which case we will skip all other tasks because it means that someone from the team has already taken care of the previous Ansible failure. However, if the lock still exists, we then try to restart the solr service but not allow Ansible to fail, so that we can remove the lock even if the restart fails. You will not allow Ansible to fail if there are multiple servers.

If there are only two servers behind the load balancer and the solr service restart fails, you should NOT remove the lock, as restarting the second server in this case can result in an outage. You can also notify the user by sending a mail asking them to troubleshoot the issue manually. We will see how mail notification works later in this chapter. Consider the following screenshot where we’re running the rollback playbook:
As explained before, Ansible checks if the lock exists, starts the service, and then removes the lock.

Instead of running the rollback playbook manually on every failure, you can write a small wrapper script, which can run the Ansible playbook, and depending on the exit code of the Ansible playbook, it can run the rollback playbook if required. It could be as simple as a bash script, which directly runs the `ansible-playbook` command.

**Callback plugins**

We now proceed to callback plugins. One of the features that Ansible provides is a callback mechanism. You can configure as many callback plugins as required. These plugins can intercept events and trigger certain actions. The section on alerting that we will cover after this section shows you how to use these plugins to make sure you get the right sort of feedback. Let’s see a simple example where we just print the run results at the end of the playbook run as part of a callback and then take a brief look at how to configure a callback.

We will rerun the `build_agent` role from the last chapter for this example; the build agent is already configured and we’re running Ansible against it. This is shown in the following screenshot:

For More Information:

As you can see in the preceding screenshot, the callback plugin resulted in an extra line called *Run Results*, and printed a dictionary or hash of the actual results. You can utilize this information in a number of ways. Isn't this powerful? Are you getting any ideas around how you can utilize a feature such as this? Do write it down before reading further. If you're able to write out even two use cases that we've not covered here and is relevant to your infrastructure, give yourself a pat on the back!

The callback plugin's location is present in *ansible.cfg*. You can use `grep` for the term "callback" in *ansible.cfg*, shown as follows:

```
$ grep callback ansible.cfg
callback_plugins   = /usr/share/ansible_plugins/callback_plugins
```

You need to then add your callback scripts in this folder. If you have five callback scripts to perform different operations, you need to add all five of them in the `callback_plugins` folder. The callbacks are in Python and you'll have to know a little bit about Python to make good use of them. You can pretty much copy and paste the following example and get started with the callback plugins.

We have a simple *callback_sample.py* file to start with, the output of which generates the *Run Results* line in the earlier example. Let's look at the folder that's configured in *ansible.cfg*, which is `/usr/share/ansible_plugins/callback_plugins` in our case:

```
$ ls -l /usr/share/ansible_plugins/callback_plugins
callback_sample.py
```

Let's now look at the contents of the *callback_sample* file:

```
$ cat /usr/share/ansible_plugins/callback_plugins/callback_sample.py

class CallbackModule(object):
    def on_any(self, *args, **kwargs):
        pass

    def runner_on_failed(self, host, res, ignore_errors=False):
        pass

    def runner_on_ok(self, host, res):
        pass
```

---

*For More Information:*

def runner_on_error(self, host, msg):
    pass

def runner_on_skipped(self, host, item=None):
    pass

def runner_on_unreachable(self, host, res):
    pass

def runner_on_no_hosts(self):
    pass

def runner_on_async_poll(self, host, res, jid, clock):
    pass

def runner_on_async_ok(self, host, res, jid):
    pass

def runner_on_async_failed(self, host, res, jid):
    pass

def playbook_on_start(self):
    pass

def playbook_on_notify(self, host, handler):
    pass

def playbook_on_no_hosts_matched(self):
    pass

def playbook_on_no_hosts_remaining(self):
    pass

def playbook_on_task_start(self, name, is_conditional):
    pass

def playbook_on_vars_prompt(self, varname, private=True, prompt=None,
                             encrypt=None, confirm=False, salt_size=None, salt=None,
                             default=None):
    pass

def playbook_on_setup(self):
    pass

def playbook_on_import_for_host(self, host, imported_file):
    pass

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Error Handling, Rollback, and Reporting

```python
pass

def playbook_on_not_import_for_host(self, host, missing_file):
    pass

def playbook_on_play_start(self, pattern):
    pass

def playbook_on_stats(self, stats):
    results = dict([(h, stats.summarize(h)) for h in stats.processed])
    print "Run Results: %s" % results
```

As you can see, the callback class, `CallbackModule`, contains several methods. The methods of this class are called and the Ansible run parameters are provided as parameters to these methods. Playbook activities can be intercepted by using these methods and relevant actions can be taken based on that. Relevant methods are called based on the action, for example, we've used the `playbook_on_stats` method (in bold) to display statistics regarding the playbook run. Let's run a basic playbook with the callback plugin and view the output as follows:

```
$ cat play.yml
---
- hosts: default
  remote_user: vagrant
  sudo: yes
  tasks:
    - name: Test message
      debug: msg="hello"

$ ansible-playbook -i inventory --private-key ~/.ssh/key.yml play.yml
```

You can now see the `Run Results` line right at the end, which is due to our custom code. This is just an example of how you can intercept methods and use them to your advantage.
However, there are so many other methods that you can utilize. Spend some time looking at the names of the methods. With all that you learned so far, you must be in a position to recognize or guess what each preceding method might do. The word `pass` indicates no action and is the default value for all these methods that you see.

If you find it difficult to understand these methods, you can print the output at the method level and that will provide more food for thought on how you can utilize these methods to your advantage. For example, let's modify the `runner_on_ok` task and print the output that we get from `host` and `res`:

```python
def runner_on_ok(self, host, res):
    print 'host - %s , res - %s' % (host,res)
```

On rerunning the playbook, it prints all the facts. A snippet of what we get with the facts is shown below:

```
L任务ING FACTS

Variety of facts associated with the remote system given to us on a platter in the form of JSON. This basically means you can even update your inventory system by just running the Ansible `setup` task against all your nodes, parse out the `res` hash or dictionary, and then make the right calls to check or update your inventory system.

(By default, if the playbook doesn't have any task, it will just run the `setup` task to get information regarding the nodes. We urge you to try out a playbook with zero tasks but with `gather_facts` not set to `False`. See the result for yourself!) Does this make sense? We strongly urge you to read through this section a couple of times to get a better understanding.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
```
For the Java tasks that we had in the playbook in our previous chapter, the output appears as shown in the following screenshot:

Looking at the preceding output, we can conclude that every module that you write in your playbook is queryable or addressable as part of the callback and you can utilize it for better reporting.

Let's see an example where we use callbacks to push information regarding the Ansible playbook run to a MySQL instance for daily, weekly, and monthly reporting, and to analyze what tasks fail the most. You will have to set up your MySQL database and make sure there is connectivity from the command center where the callbacks are being configured. It's time to look at the plugin code, which is as follows:

```
$ cat callback_mysql.py (remember this callback function should be in the 'callback_plugins' folder as indicated in ansible.cfg file)
import getpass
import MySQLdb as mdb
import os

class CallbackModule(object):
    
    def __init__(self):
        self.dbhost = os.getenv('DBHOST')
        self.dbuser = os.getenv('DBUSER')
        self.dbpassword = os.getenv('DBPASSWORD')
        self.dbname = os.getenv('DBNAME')
        self.action = ""
        self.user = ""

    def update_stats(self, host, result, task=None, message=None):
        con = mdb.connect(self.dbhost, self.dbuser, self.dbpassword, self.dbname)
        cur = con.cursor()
        cur.execute('insert into ansible_run_stats(user,host,result,task,message) values("%s","%s","%s","%s","%s")' % (self.user, host, result, task, message))
```

For More Information:

www.packtpub.com/networking-and-servers/learning-ansible
con.commit()
con.close()

def on_any(self, *args, **kwargs):
    pass

def runner_on_failed(self, host, res, ignore_errors=False):
    self.update_stats(self.task.play.hosts, "unsuccessful", task=self.
    task.name, message=res)

def runner_on_ok(self, host, res):
    pass

def runner_on_error(self, host, msg):
    pass

def runner_on_skipped(self, host, item=None):
    pass

def runner_on_unreachable(self, host, res):
    self.update_stats(self.task.play.hosts, "unsuccessful", task=self.
    task.name, message=res)

def runner_on_no_hosts(self):
    pass

def runner_on_async_poll(self, host, res, jid, clock):
    pass

def runner_on_async_ok(self, host, res, jid):
    pass

def runner_on_async_failed(self, host, res, jid):
    pass

def playbook_on_start(self):
    pass

def playbook_on_notify(self, host, handler):
    pass

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
def playbook_on_no_hosts_matched(self):
    pass

def playbook_on_no_hosts_remaining(self):
    pass

def playbook_on_task_start(self, name, is_conditional):
    pass

def playbook_on_vars_prompt(self, varname, private=True, prompt=None,
                           encrypt=None, confirm=False, salt_size=None, salt=None, default=None):
    pass

def playbook_on_setup(self):
    pass

def playbook_on_import_for_host(self, host, imported_file):
    pass

def playbook_on_not_import_for_host(self, host, missing_file):
    pass

def playbook_on_play_start(self, pattern):
    if not self.user:
        self.user = getpass.getuser()

def playbook_on_stats(self, stats):
    if not stats.dark and not stats.failures:
        self.update_stats(stats.ok.keys()[0], "successful")

Let's look at the code in more detail to understand the flow; we first initialize
the database parameters by getting them from the environment in the following
snippet. We expect the following environment variables to be set: DBHOST, DBUSER,
DBPASSWORD, and DBNAME:

def __init__(self):
    self.dbhost = os.getenv('DBHOST')
    self.dbuser = os.getenv('DBUSER')
    self.dbpassword = os.getenv('DBPASSWORD')
    self.dbname = os.getenv('DBNAME')
    self.action = ""
    self.user = ""

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Since we might have different users who might log in with their account onto the system, we need to make sure we trace who runs what commands. We store the user value in the following method:

```python
def playbook_on_play_start(self, pattern):
    if not self.user:
        self.user = getpass.getuser()
```

We then track the status of the playbook run by calling the `update_stats` method (this can either be a class instance method or a method that's outside the class) from the `runner_on_failed`, `runner_on_unreachable`, and `playbook_on_stats` methods to capture the status of the playbook run as follows:

```python
def runner_on_failed(self, host, res, ignore_errors=False):
    self.update_stats(self.task.play.hosts, "unsuccessful", task=self.task.name, message=res)

def runner_on_unreachable(self, host, res):
    self.update_stats(self.task.play.hosts, "unsuccessful", task=self.task.name, message=res)

def playbook_on_stats(self, stats):
    if not stats.dark and not stats.failures:
        self.update_stats(stats.ok.keys()[0], "successful")
```

Finally, the `update_stats` method connects to the database and pushes the data into a table called `ansible_run_stats`, shown as follows:

```python
def update_stats(self, host, result, task=None, message=None):
    con = mdb.connect(self.dbhost, self.dbuser, self.dbpassword, self.dbname)
    cur = con.cursor()
    cur.execute('insert into ansible_run_stats(user,host,result,task,message) values("%s","%s","%s","%s","%s")' % (self.user, host, result, task, message))
    con.commit()
    con.close()
```

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Let's look at the rows in the database after a few runs. The output is shown in the following screenshot:

As expected, Ansible recorded all the passed and failed runs' details in the MySQL database. This is a great way to report. Make sure that you dump all the information related to all your runs across all users and run reporting scripts either as cron jobs or from other BI systems to send out reports with information on the number of users, hosts, and tasks that run across your infrastructure using Ansible.

Let's look at the other possible use cases as follows:

- Let's say you're starting a deployment using Ansible and you want to notify a bunch of people about it. In that case, you can utilize the `playbook_on_start` method to alert all of them.
- You can change the way the log output looks for all the tasks.
- You can calculate the overall runtime taken by storing the start time and end time as instance variables.
- You can alert anyone based on each of the outputs. We'll see a couple of examples in the next section

### Monitoring and alerting

Monitoring and alerting go hand in hand. At an infrastructure and application level, you end up monitoring several metrics, right from CPU and memory to application-level information such as heap and number of database connections from your application, and alert relevant teams based on that. Also, from an infrastructure automation point of view, you need to make sure you start a strong feedback loop by integrating automation with your monitoring and alerting system or systems through constant reporting.
We’ll focus on how you can keep on the Ansible run for any failures or confirmation of task completions rather than staring at your screen continuously. At some stage, you might run all the tasks from a custom UI. It should be fine if you are running a small playbook with a bunch of tasks. However, if it’s a long playbook with many tasks, you might derail the flow and miss some of the errors or an important task because of too much data popping out of your stdout output. To deal with such a situation, we use **Alerting**. You can notify the user whenever a task/playbook fails or after a task is completed, allowing them to sip their brew, sit back, and relax. There are different types of alerting systems, such as mail, monitoring systems, graphing systems, pagers, chat rooms, and so on. We will look at some of the basic alerting techniques that can be used with Ansible, which are as follows:

- **E-mails**
- **HipChat**
- **Nagios**
- **Graphite**

### E-mails

The easiest and most common way of alerting is to send e-mails. Ansible allows you to send e-mails from your playbook using a mail module. You can use this module in between any of your tasks and notify your user whenever required. Also, in some cases, you cannot automate each and every thing because either you lack the authority or it requires some manual checking and confirmation. If this is the case, you can notify the responsible user that Ansible has done its job and it’s time for him/her to perform his/her duty. Let’s see how you can use the mail module to notify your users with one of the example playbooks of *Chapter 3, Taking Ansible to Production*, as follows:

```
$ std.run/playbooks/mail_alert.yml
- hosts: web01
  tasks:
    - name: test user
      user:
        name: ansible
        state: present
      register: result
    - name: Set database
      mysql_database:
        name: database
        user: ansible
        state: present
        host: localhost
        database: db
        port: 3306
      register: result
    - name: Set user
      mysql_user:
        name: ansible
        state: present
        host: localhost
        database: db
        port: 3306
        priv: all
        password: ansible
        user: ansible
      register: result
    - name: Set database
      mysql_database:
        name: database
        state: present
        host: localhost
        database: db
        port: 3306
        priv: all
        password: ansible
        user: ansible
      register: result
    - name: Give users access to multiple databases
      mysql_user:
        name: ansible
        state: present
        host: localhost
        database: db
        port: 3306
        priv: ['all', 'show', 'process'],
        password: ansible
        user: ansible
      register: result
    - name: Sending mail to user
      mail:
        host: localhost
        port: 25
        from: root@localhost
        to: root@localhost
        subject: "Ansible Alert"
        body: "Hello, ansible has done its job and it's time for you to perform your duty."
```

For More Information:

In the preceding playbook, we will first loop over the user and database dictionary giving the MySQL users access to the database. If the tasks succeed, we will send an e-mail to the user saying all tasks were completed successfully. Let's see how this works in the following screenshot:

As expected, Ansible tried giving database access to the users, but it found that the users already had access to the database, thus returning with an **OK** status instead of a **CHANGED** status. Let's see if they actually received an e-mail. The following screenshot shows an e-mail was sent to the user:

Bingo! It worked. Likewise, you can also send the stdin of a task by using variables within the body of your e-mail.

You can also use this module before a prompt task. For example, let's say that you have a playbook that takes around 20–30 minutes to complete, and at the end of the playbook, you ask your user (via a command prompt), whether Ansible should add the host back to the load balancer, since you cannot expect a user to sit in front of a screen for 30 minutes and keep waiting for the prompt to respond. Instead, you can e-mail your users before prompting so that they can get back to the Ansible prompt and do the needful.

For More Information:

**HipChat**

**HipChat** is a private group chat and IM network that provides real-time collaboration features for companies and teams. Its usage has grown heavily over the last year or so. Ansible allows you to send notifications to a HipChat room from a playbook using a `hipchat` module. To send a notification, you need to first create a token for the room where you want to notify.

Refer to the **HipChat** documentation to create API tokens.

Let's see how you can use the `hipchat` module to notify users, as shown in the following screenshot:

![HipChat notification](image_url)

In the preceding example, we reused the database playbook, which we saw in the *E-mails* section of this chapter. We pass the `hipchat` API token, room name, message, and sender name to the `hipchat` module.

Note that the `hipchat` module does not support HipChat v2 APIs at the time of writing. You should use a HipChat v1 API with this module.

For More Information:
Error Handling, Rollback, and Reporting

Let's run the playbook. The output is shown in the following screenshot:

```
$ ansible-playbook -i hosts hipchat_alert.yml
PLAY [web001] *******************************************************
TASK: [give users access to multiple databases] *******************************************************
ok: [web001] => (item={"name": 'alice', 'clientdb'})
ok: [web001] => (item={"name": 'alice', 'employeedb'})
ok: [web001] => (item={"name": 'alice', 'providerdb'})
ok: [web001] => (item={"name": 'bob', 'clientdb'})
TASK: [Sending mail to user] *******************************************************
ok: [web001]
TASK: [Notifying on hipchat] *******************************************************
changed: [web001]
PLAY RECAP *******************************************************
web001          : ok=3  changed=1  unreachable=0  failed=0
```

As expected, Ansible gave the database access to the user, sent an e-mail to the user, and notified in the HipChat room. Let's see if we actually received a notification in the ops room that we configured. We should receive something like the following screenshot:

![HipChat notification](image)

Yes we did! We received a notification in our HipChat room. Wasn't that quite easy? Now, it's your job to try out this awesome module.

Nagios

No book or chapter on monitoring or alerting is complete without talking about Nagios. Nagios is a widely used, open source monitoring tool, which allows you to monitor almost everything that exists on planet Earth and maybe beyond. Well, jokes apart, you can use Nagios to monitor your playbook run by sending passive checks to the nagios server. As we did for the e-mail notifications, you can have a nagios task at the end of a playbook, which will notify the nagios server if a playbook succeeds; alternatively, you can have a task in the rollback playbook, which will notify the nagios server that a playbook has failed.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
There are other popular monitoring tools such as Zabbix, Ganglia, and Sensu. You can integrate Ansible in a way similar to what we've shown with Nagios in this example.

Let's see an example playbook that uses a MySQL database and notifies our `nagios` server if everything goes well.

In the preceding playbook, we first dump the MySQL table using the `mysqldump` command. If it goes well, we then use a passive check to notify the `nagios` server using the `send_nsca` command. These kinds of backup-and-restore tasks are simple to perform but very often teams forget to do them or do not have the right kind of alerting or monitoring around them and end up wasting a lot of time on them. We felt it's best to show how you can do something as simple as this with Ansible, and at the same time, stress on important best practices such as this in your organization. If you're already doing it, great!

Let's run the playbook. We expect to see an `ok` output in Nagios at the end of the playbook run, as shown in the following screenshot:

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
As expected, Ansible successfully performed the MySQL dump of the Ansible database and notified the nagios server. Let's see if the check is updated on our nagios monitoring.

Hey, it worked! This was a simple example of Nagios integration to get you started.

Another example of Nagios alerting is to use it with some search indexing. For example, you perform an indexing of your site every midnight and want to monitor this using Nagios. To perform this activity, you can write a playbook, which will index your data and make sure that the production traffic is not affected because of this activity. At the end of the playbook, that is, the last task, you will notify your nagios server about the completion of the activity using a passive check. We'll leave this as an exercise for you to try out.

You can also add the send_nsca command to the Ansible callback and notify Nagios on each failed, unreachable, or ok status. Let's see how a callback script for this activity would look:

```python
$ cat callback_nagios.py
import subprocess

class CallbackModule(object):
    def __init__(self):
        self.play_name = ""

    def nagios_passive_check(self, host, return_code, status):
        subprocess.call("echo -e '%s\ts\t%d\t%s' | sudo /usr/sbin/send_nsca -H 10.47.137.69 -c /etc/nagios/send_nsca.cfg" % (host, self.play_name, return_code, status), shell=True)

    def runner_on_failed(self, host, res, ignore_errors=False):
        self.nagios_passive_check(host, 2, "Critical: %s" % res)

    def runner_on_unreachable(self, host, res):
        self.nagios_passive_check(host, 2, "Critical: %s" % res)

    def playbook_on_play_start(self, pattern):
        self.play_name = self.play.name

    def playbook_on_stats(self, stats):
        if not stats.dark and not stats.failures:
            self.nagios_passive_check(stats.ok.keys()[0], 0, "Ok: Successful")
```

For More Information:

www.packtpub.com/networking-and-servers/learning-ansible
Let's look at the flow of the code in more detail as follows:

1. We initialize a variable, `play_name`, with an empty value as follows. The intent is to store the playbook name in this variable so that we can access it from any of the methods within the `CallbackModule` class:
   ```python
def __init__(self):
    self.play_name = ""
```

2. We create a method to actually call the `send_nsca` command and update the `nagios` server as follows. We will pass the host against which Ansible is running, `return_code` for the `nagios` server, and finally the message, that is, the status of the playbook for the `nagios` server:
   ```python
def nagios_passive_check(self, host, return_code, status):
    subprocess.call("echo -e '%s\t%s\t%d\t%s' | sudo
     /usr/sbin/send_nsca -H 10.47.137.69 -c
     /etc/nagios/send_nsca.cfg" % (host,
      self.play_name, return_code, status),
      shell=True)
```

3. We assign the play name from the `playbook_on_stats` method. Make sure you add the correct play name to your playbook because we will be using this play name as a service name in nagios:
   ```python
def playbook_on_play_start(self, pattern):
    self.play_name = self.play.name
```

4. We call the `nagios_passive_check` method from the `runner_on_failed`, `runner_on_unreachable`, and `playbook_on_stats` methods. This method will notify the `nagios` server on receiving the failure, unreachable, and ok statuses, respectively. The `stats.dark` method in the third method, shown as follows, means the status was not unreachable, whereas `stats.failures` means the status was not failure:
   ```python
def runner_on_failed(self, host, res, ignore_errors=False):
    self.nagios_passive_check(host, 2, "Critical: %s" % res)

def runner_on_unreachable(self, host, res):
    self.nagios_passive_check(host, 2, "Critical: %s" % res)

def playbook_on_stats(self, stats):
    if not stats.dark and not stats.failures:
        self.nagios_passive_check(stats.ok.keys()[0], 0,
         "Ok: Successful")
```

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Graphite

Graphite is yet another extensively used tool in operations for real-time graphing. Graphite thrives when it comes to time series. You can read more about Graphite at http://graphite.readthedocs.org/en/latest/index.html.

To use Graphite, you need to send the following three parameters to it:

- The metric name
- Value of the metric
- Timestamp

Graphite provides an easy API that allows you to alert a user using a tool such as Nagios, Cabot, or Riemann. You can read more about Graphite at http://graphite.wikidot.com.

Now, the question is, what is it that we'd like to track from an Ansible perspective in Graphite? Typically, as a best practice from a configuration management perspective, we'd like to know how many tasks changed during an Ansible run. We'd also like to know whether the Ansible run itself passed or failed. If it failed, did it fail because the node was unreachable or because there was an issue with the playbook?

We'll do all of these from a Graphite plugin. We've seen how to check for status with Nagios. We'll go one step further by also monitoring the number of tasks and how many actually ran in each run. Let's see the code first, which is as follows:

```python
$cat graphite_plugin.py

import socket
import time

class CallbackModule(object):

def __init__(self):
    self.UNREACHABLE_RUN = 0
    self.FAILED_RUN = 25
    self.SUCCESSFUL_RUN = 50
    self.CARBON_SERVER = '192.168.1.3'
    self.CARBON_PORT = 2003
    self.playbook = ''

    # Process the run result for each host
    def process_result(self, res):
        status = self.SUCCESSFUL_RUN

        # [176]

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
changed_tasks, tasks = 0,0
if type(res) == type(dict()):
    for key in res:
        host = key
        if res[host]['unreachable'] == 1:
            status = self.UNREACHABLE_RUN
        elif res[host]['failures'] != 0:
            status = self.FAILED_RUN
        else:
            tasks = res[host]['ok'] + res[host]['changed']
            changed_tasks = res[host]['changed']
host = host.replace('.','-')
send_data_to_graphite(host,status,tasks,changed_tasks)

def send_data_to_graphite(self, host, status, tasks, changed_tasks):
    prefix = "ansible.run.%s.%s" % (self.playbook,host)
    tasks_metric = "%s.number_of_tasks %d %d\n" % (prefix,tasks,int(time.time()))
    status_metric = "%s.status %d %d\n" % (prefix,status,int(time.time()))
    changed_tasks_metric = "%s.changed_tasks %d %d\n" % (prefix,changed_tasks,int(time.time()))
    print "Prefix", prefix
    print "Tasks: %d, status: %d, changed_tasks: %s" % (tasks,status,changed_tasks)
    sock = socket.socket()
    sock.connect((self.CARBON_SERVER, self.CARBON_PORT))
    sock.sendall(status_metric)
    sock.sendall(tasks_metric)
    sock.sendall(changed_tasks_metric)
    sock.close()

## Other methods in the plugin do not change ##

def playbook_on_play_start(self, pattern):
    self.playbook = pattern

def playbook_on_stats(self, stats):
    results = dict([(h, stats.summarize(h)) for h in stats.processed])
    self.process_result(results)
Here, we see that we change only two methods in the callback:

- `playbook_on_play_start`: This method is used in order to store the name of the play
- `playbook_on_stats`: This method is used to get the final stats of the playbook run

The metric values that we'll send to Graphite are in the `send_data_to_graphite` method as follows:

- **Metric for the overall number of tasks:** (syntax: `ansible.run.<playbook name>.<hostname>.number_of_tasks`)
  To create a hierarchy in Graphite, you need to separate the metric names by using a dot. Hence, as part of this exercise, we replaced the dots in an IP address with a hyphen.

- **Status of the run:** (syntax: `ansible.run.<playbook name>.<hostname>.status`)
  As part of the status, we've covered three states: `SUCCESSFUL_RUN`, `FAILED_RUN`, and `UNREACHABLE_RUN`. Each of the three statuses has specific values that you can easily map out in Graphite: 50, 25, and 0. This is up to you to customize.

- **Number of changed tasks:** (syntax: `ansible.run.<playbook name>.<hostname>.changed_tasks`)

Make sure you have copied the graphite callback in the callbacks directory. We again use our `build_agent` role from the previous chapter to demonstrate this example. We bring up a new node and run the playbook. It results in the following output at the end of the run:

```
PLAY RECAP ****************************************************
Prefix ansible.run.build_agents.192-168-33-11
Tasks: 18, status: 50, changed_tasks: 8
192.168.33.11  : ok=10  changed=8  unreachable=0  failed=0
```

Overall, 18 tasks ran and eight changed. Downloading of files does not change the state of the system and hence does not have the `changed` status associated with them.
Let's run these tasks again. We assume here that you run Ansible on each of the machines every hour, to maintain the state, or at the start of every build. We obviously haven't run the task every hour, because we want to show you what graphs are going to look like (We ran it within 5 minutes). So coming back, on rerunning, we find that eight tasks have run and none of the tasks have failed, as shown in the following screenshot:

You might be wondering how the jump from 18 to 8 happened. This is because the playbook has 10 tasks that depend on conditionals. For example, the task will download Maven if the directory is not present. Hence, all those tasks are skipped. You can look at the build_agent playbook role in the previous chapter to go over the playbook content again.

**Time for an error**

Now, we'll introduce an error in `ant.yml` and rerun the playbook. The output is shown as follows:

The Ansible run status will be set to **FAILED_RUN** from **SUCCESSFUL_RUN** and you will see a change in Graphite if everything works as expected.
We then fix the error and rerun the playbook, and we again see eight tasks. Let’s see the snapshots in Graphite spread across a 10-minute period that captures all these runs.

![Graphite Composer](image)

You can see the three metrics that we’ve chosen along with their respective values and how they’ve changed. On the left-hand side pane, you’ll be able to see the hierarchy that we’ve introduced as follows:

![Hierarchy](image)

Now, we remove the status to compare just the overall number of tasks and the number of changed tasks.

For More Information:
www.packtpub.com/networking-and-servers/learning-ansible
Over a period of time, if there is no change, you should have unchanged values for the `number_of_tasks` and `changed_tasks` metrics.

On the Graphite server filesystem, you will be able to see the following hierarchy for the three metrics that we measured:

```
[root@node build_agents]# pwd
/var/lib/carbon/whisper/ansible/run/build_agents
[root@node build_agents]# tree
  192-168-33-11
    |--- changed_tasks.wsp
    |--- number_of_tasks.wsp
    |--- status.wsp
1 directory, 3 files
[root@node build_agents]# whisper-fetch 192-168-33-11/changed_tasks.wsp | grep -v None
1407691680 8.000000
1407691740 0.000000
1407691860 0.000000
1407691980 0.000000
1407692040 0.000000
[root@node build_agents]#
```
Error Handling, Rollback, and Reporting

You can see the directory from where we ran the `tree` command. Graphite stores all the data in the form of whisper files. You can query the values of each file on the filesystem using `whisper-fetch`, as shown in the preceding screenshot. You can also run a `curl` call to obtain the values in the JSON format. A `curl` call to obtain information about the last 30 minutes and in the JSON format should look like the following:

```
$ curl "http://<graphite ip>/render?target=ansible.run.build_agents.192-168-33-11.*&format=json&from=-30min"
```

The following is the output of the preceding command line:

The Graphite website has a great, detailed documentation on how to consume the data from Graphite. We’ll leave it to you to figure out the rest! While consuming the Graphite metrics, let’s say from Nagios, you can set up an alert that will filter out all the null values and alerts based on either the failure value or the lack of idempotency in your playbooks.

For More Information:

www.packtpub.com/networking-and-servers/learning-ansible
Summary

With this, we come to the end of this chapter where we’ve seen how to handle errors using rollbacks, how to use callbacks in general, and how to report and alert using tasks as well as callbacks. Along the way, we’ve also seen other tools such as etcd, MySQL database, HipChat, E-mail, Nagios, and Graphite, and how to integrate Ansible with them. These are definitely some common as well as easy implementations that provide great value to Ansible users. Finally, it’s very important that you have a strong feedback loop in your infrastructure and you’ve seen a few of those techniques in action in this chapter.

It’s also time for a coffee break. As usual, we have a few questions for you to ponder over and discuss with your team as follows:

- What monitoring system do you have and what features from this chapter would benefit you the most?
- If you use callbacks, what would you want your callbacks to do?
- If you want to alert, about what would you alert?

We hope that you’ll revisit the roles that you’ve written, or started to write, after reading Chapter 3, Taking Ansible to Production, and incorporate some of the Ansible features that you’ve learned in this chapter to introduce feedback loops into your infrastructure automation. If you’re a sysadmin, we can imagine you smiling after reading through this chapter!

The next chapter is all about writing custom modules and testing them. This will test some of your basic programming skills. So get ready to code.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.