Next Steps with Ansible

In this, our final chapter, we will discuss how you can integrate Ansible into your day-to-day workflows. We will cover continuous integration tools, monitoring tools, and troubleshooting.

We will discuss the following topics:

* Integrating with third-party services
* How you can use Ansible to troubleshoot problems when they occur
* Some real-world examples

Let’s dive straight in and look at how we can hook our playbooks into third-party services.

Technical requirements

This chapter will differ from previous ones. While code examples are given in the chapter and the GitHub repository, they will not be complete working examples. Instead, we will discuss integrating them into your projects so they are more of the art of the possible rather than fully formed examples.

Integrating with third-party services

Although you may be running the playbooks yourself, it’s a good idea to keep a log of your playbook run or update other team members or departments with the results. Ansible has several modules that allow you to work with third-party services to provide real-time notifications.

Let’s start by looking at Slack.

Slack

Slack has rapidly become the preferred option for team-based collaboration services across different IT departments. One key benefit of Slack is its support for third-party applications via its App Directory; Ansible supports Slack Incoming Webhooks via the community.general.slack module.

Remember, you can install the community.general collection if you don’t have it installed by running the following command:

$ ansible-galaxy collection install community.general

CopyExplain

Before we look at the Ansible code, we should quickly discuss how you create a Slack App and enable webhooks.

First, you must make your own Slack app; you can do this by visiting https://api.slack.com/apps/new. Once there, click the **Create an App** button and select the **From Scratch** option. From here, you need to fill in the **App Name** and **Pick a workspace to develop your app in**, which for the majority of us will be your primary workspace, as you can see from the following screenshot:

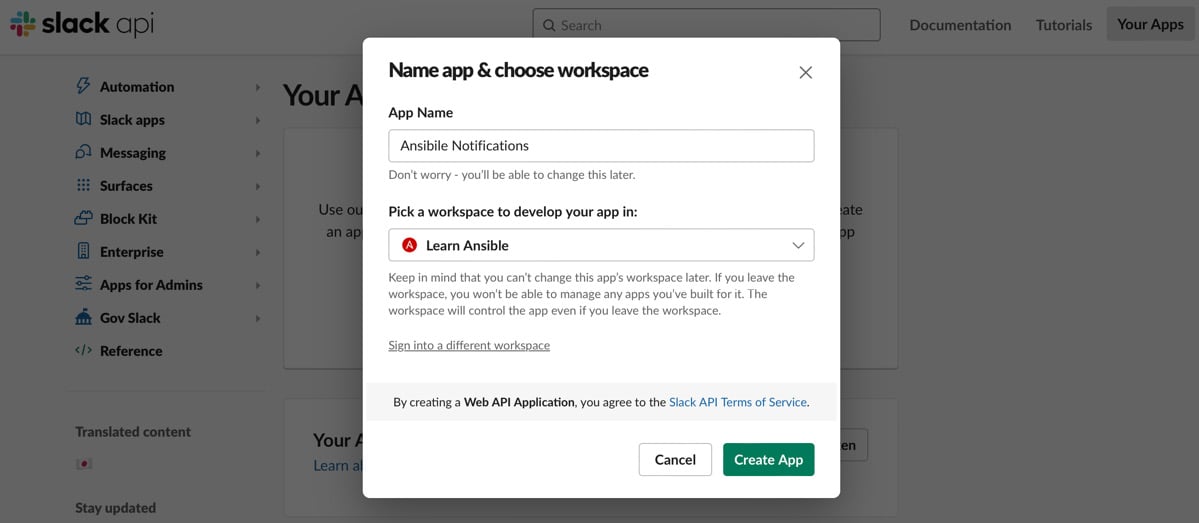


Figure 17.1 – Creating the Slack app

Once the Slack App has been created, you will be taken to your new application settings page. In the left-hand menu, you should see an option for **Incoming Webhooks**. Go to this page and toggle the **Activate Incoming Webhooks** switch to **On**. This will extend the options and give you the option to **Add New Webhook to Workspace**.

From here, you will need to select where you would like your Slack App to post; as you can see from the following screenshot, I selected the **#general** channel:

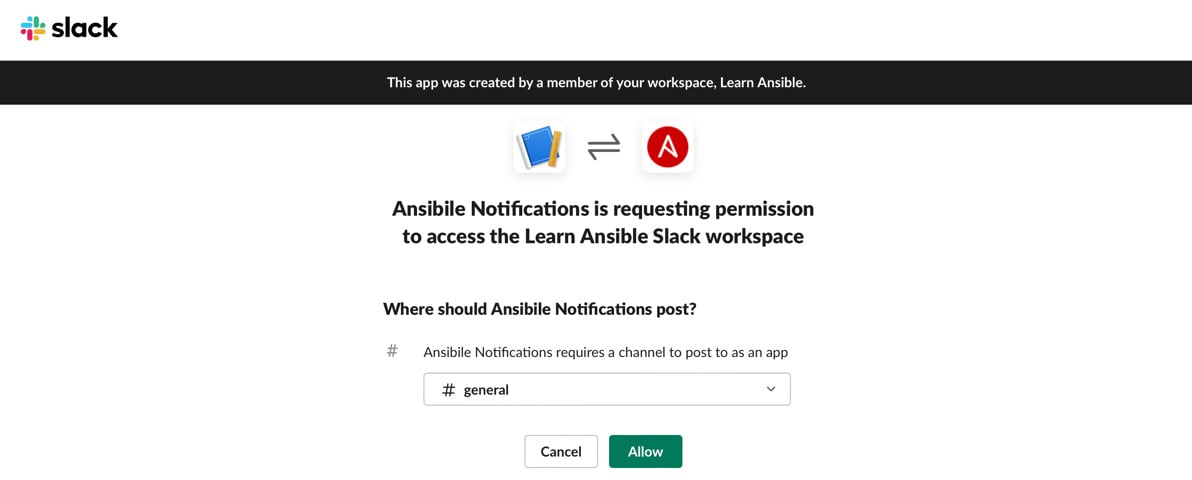


Figure 17.2 – Choosing where to post

Once selected, you will be taken back to the **Incoming Webhooks** page for your application; here, you will be given a Webhook URL, which should look something like the following, and you will need to make a note of this and keep it safe (the following one has been revoked):

https://hooks.slack.com/services/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

CopyExplain

Now that we have everything we need to interact with Slack, we can examine the code. As mentioned at the start of the chapter, I will only go into some of the code, as much of it will already be familiar.

There is just a single variable we need to add, and it is the token used to identify and authenticate against the webhook we created: the token is everything after https://hooks.slack.com/services/ in the webhook URL from Slack, so in my case, the variable, which I put in group\_vars/common.yml, looks like this:

slack:

  token: "TBCRVDMGA/B06RCMPD6R4/YBTo7ZXZHrRg57fvJXr1sg43"

CopyExplain

As this token should be treated as a secret, I recommend also using Ansible Vault to encrypt the value, so to do this, you can run the following:

$ ansible-vault encrypt\_string 'TBCRVDMGA/B06RCMPD6R4/YBTo7ZXZHrRg57fvJXr1sg43' --name 'token'

CopyExplain

The token in the repo is encrypted using Ansible Vault, and as it has been revoked, you will need to update it with your own.

By jumping straight into roles/slack/tasks/main.yml, you can see that the playbook launches a resource group, virtual network, and subnet in Azure.

There are no changes to the first tasks that launch the Azure resources:

- name: "Create the resource group"

  azure.azcollection.azure\_rm\_resourcegroup:

    name: "{{ resource\_group\_name }}"

    location: "{{ location }}"

    tags: "{{ common\_tags }}"

  register: "resource\_group\_output"

CopyExplain

Additionally, the debug task we used in previous chapters is still there; immediately after the debug task, we have the task (well, sort of) which sends the notification to Slack:

- name: "Notify the team on Slack about the resource group status"

  include\_tasks: slack\_notify\_generic.yml

CopyExplain

As you can see, it triggers another task in the slack\_notify\_generic.yml file, and we pass the registered output’s content as a set of variables, most of the them are self-explanatory:

  vars:

    resource\_changed: "{{ resource\_group\_output.changed }}"

    resource\_type: "Resource Group"

    resource\_name: "{{ resource\_group\_output.state.name }}"

    resource\_location: "{{ resource\_group\_output.state.location }}"

CopyExplain

The last two are a little different; this one takes the full resource ID and prefixes it with https://portal.azure.com/#resource, as the resource ID is the URL for the resource in Azure; this, together with the URL prefix, will give us a clickable link that will take the user directly to the resource when they follow it:

    azure\_portal\_link: "https://portal.azure.com/#resource{{ resource\_group\_output.state.id }}"

CopyExplain

The final variable generates a comma-separated list of tags and values using a Jinja2 template function:

    resource\_tags: >

      {% for key, value in resource\_group\_output.state.tags.items() %}

      \*{{ key }}:\* {{ value }}{% if not loop.last %}, {% endif %}

      {% endfor %}

CopyExplain

You might also have noticed that the {{ key }} variable has a \* on either side; this is not part of the template function; this is the markdown syntax for **bold**, and it will style the contents as such.

Before we look at what is in roles/slack/tasks/ slack\_notify\_generic.yml, let’s quickly discuss why we are taking this approach.

As we mentioned several times in the title, one of the main goals of automating our deployments is to streamline everything as much as possible. In this case, the task we are calling will be the standard throughout the playbook, and the only changes we need to make are the content.

So rather than repeating the community.general.slack task several times in our playbook, we can define it once and then call it multiple times. This means if we need to change something in the community.general.slack task, we only have to update it in one place.

The task itself has a little bit of logic added, so let’s review that now:

- name: "Notify the team on Slack about resource changes"

  community.general.slack:

    token: "{{ slack.token }}"

    parse: "none"

CopyExplain

As you can see from the preceding code, we are passing our webhook token and setting the parse option to none. This means that community.general.slack will not touch any content we post to the webhook to strip out formatting, etc.

Rather than sending a simple message, we use the attachments type. This will nicely format our message into blocks, and we can also set a status color based on whether there has been a change to the content or not:

    attachments:

      - fallback: "Notification about Azure resource changes"

CopyExplain

The logic for setting the color is as follows: here, we use the Boolean value of true or false that is passed by the resource\_changed variable. If the variable equals true, it means that the resource has been changed, so we set the color to the pre-defined warning color, which is orange; otherwise, the color is set to good, which is green:

        color: "{% if resource\_changed %}warning{% else %}good{% endif %}"

        title: "Ansible: {{ resource\_type }}"

CopyExplain

Next, we have the message content: here, we are using a similar logic as we did for setting the color based on whether there has been a change to the resource or not:

        text: "{{ resource\_name }} has been {% if resource\_changed %}created/updated{% else %}checked (no changes){% endif %}."

CopyExplain

Finally, we have the fields; each of these displays the information we are passing to the task in a block, apart from one:

        fields:

          - title: "Location"

            value: "{{ resource\_location }}"

            short: true

          - title: "Azure Portal"

            value: "<{{ azure\_portal\_link }}|View in Azure Portal>"

            short: true

          - title: "Tags"

            value: "{{ resource\_tags }}"

            short: false

CopyExplain

The value of the Azure portal link is a little different; Slack uses **mrkdwn**, a markup language similar to Markdown but with some differences, especially regarding formatting links. As you can see, we are setting this to the following:

<{{ azure\_portal\_link }}|View in Azure Portal>

CopyExplain

This is the mrkdwn syntax for creating a clickable link. It will link to the URL being passed in the {{ azure\_portal\_link }} variable. The text after the pipe | is the visible text that will appear in the Slack message and act as the clickable link.

When Slack renders this message, it will display **View in Azure Portal** as clickable text. When someone clicks on it, Slack will open the URL in the {{ azure\_portal\_link }} variable, directing the user to the Azure Portal.

Now that we know what the playbook looks like, let’s run it:

$ ansible-playbook -i hosts site.yml --ask-vault-pass

CopyExplain

This will prompt you to provide a valuable password and then deploy the resources; in this case, we don’t need to know the output of running the playbook and should, instead, turn our attention to Slack itself:

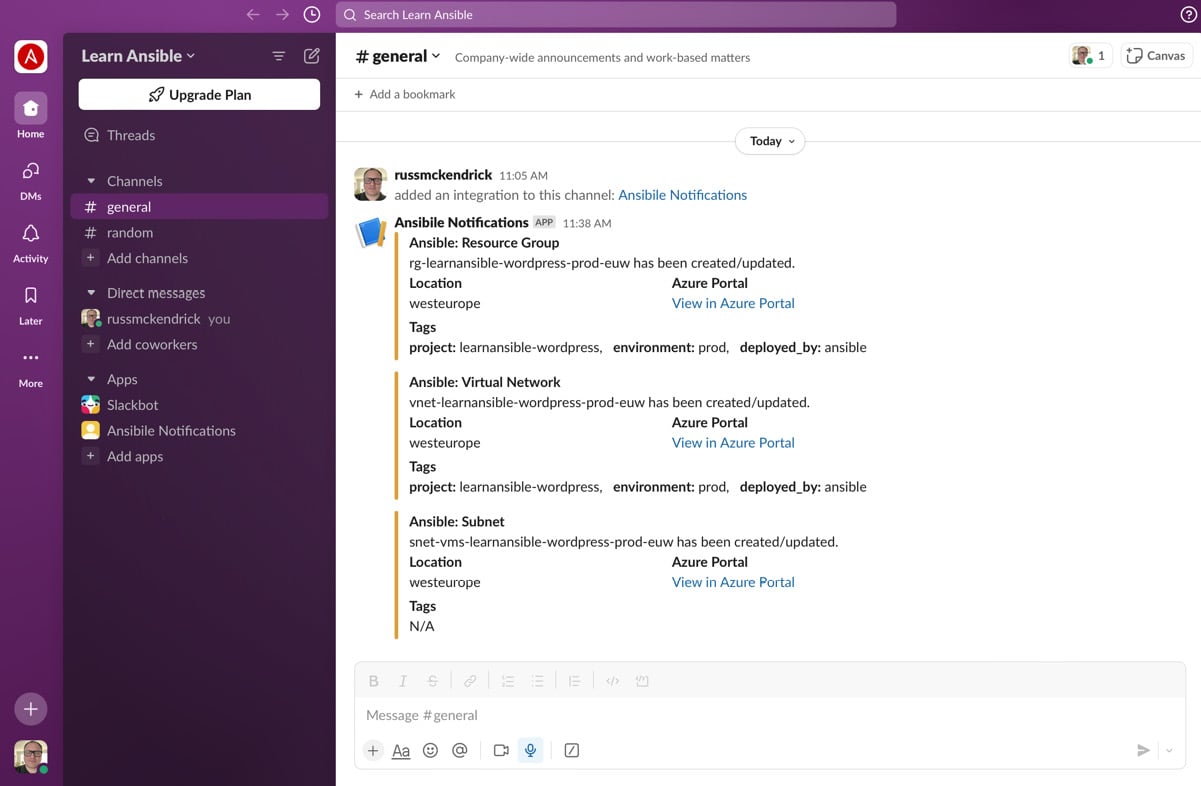


Figure 17.3 – First run of the playbook

As you can see from the preceding output, three resources have been added, so they are referred to as created/updated. The orange bar is on the left-hand side of the message.

Let’s now rerun the playbook using the following:

$ ansible-playbook -i hosts site.yml --ask-vault-pass

CopyExplain

You will see that the message now looks like this:

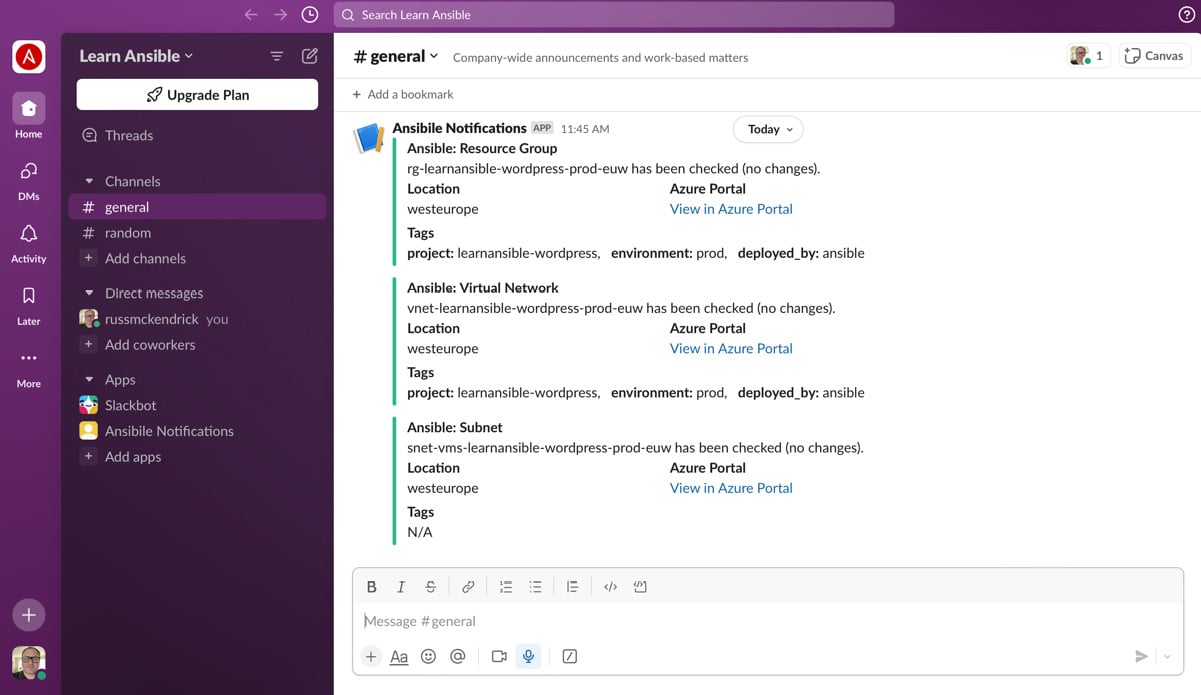


Figure 17.4 – Running the playbook a second time

This time, there have been no changes, which the message reflects. The status is also showing green, so we can quickly see that there have been no changes.

The only thing I would add is that if you look at the code in the repo, you will notice that for the subnet, we are having to make some allowances:

* resource\_location: subnets don’t have a location, so we are using the one from the virtual network the subnet is being created in
* azure\_portal\_link: while an ID for the subnet is being returned, it doesn’t precisely match the logic we use to open the resource directly in the Azure portal, so we link to the virtual network where the subnet is configured
* resource\_tags: you can’t add tags to a subnet, so we set the value to N/A

As you can see from the screens, this is useful for notifying others that your playbook is being run. It also gives you quick access to the resources being created/updated or checked and an audit trail of changes being made to your resources.

While the code we discussed only applies to Slack and the resources deployed in Microsoft Azure, the concept should apply to any integration supported by Ansible.

Other integrations

Dozens of other integrations, both community- and vendor-supported, are available on Ansible Galaxy. If you can’t find one for your use case and your target service has an API, you could quite quickly build an integration using the ansible.builtin.uri module, which is designed to interact with web APIs and services.

What follows are some example use cases for other integration modules.

Say

Most modern computers come with some level of voice synthesis built in; by using this module, you can have Ansible verbally inform you of the status of your playbook run:

    - name: "Speak an update"

      community.general.say:

        msg: "Hello from Ansible running on {{ inventory\_hostname }}"

        voice: "Zarvox"

      delegate\_to: localhost

CopyExplain

While this is fun, it isn’t very useful and could quickly become annoying, so let’s move on.

Syslog

Suppose you ship the log files from your target hosts. In that case, you may want to send the results of the playbook run to your target host machine syslog so that it is shipped to your central logging service for use in external services such as an **SIEM**, which stands for **security information and event management**, product:

- name: "Send a message to the hosts syslog"

  community.general.syslogger:

    msg: "The task has completed and all is well"

    priority: "info"

    facility: "daemon"

    log\_pid: "true"

CopyExplain

This is a great way to register that something has happened on the target host in a way that logs it along with everything else that is happening on the target operating system.

ServiceNow

ServiceNow is enterprise-grade IT service management software from ServiceNow, Inc.

By using the servicenow.servicenow.snow\_record module, your playbook can open incidents within your ServiceNow installation:

- name: "Create an incident in SNOW"

  servicenow.servicenow.snow\_record:

    username: "{{ snow.username}}"

    password: "{{ snow.passord}}"

    instance: "{{ snow.instance }}"

    state: "present"

    data:

      short\_description: "Ansible playbook run on {{ inventory\_hostname }}"

      severity: 3

      priority: 2

  register: snow\_incident\_result

CopyExplain

Once open, you can then add notes to them using something like the following:

- name: "Update the SNOW incident with work notes"

  servicenow.servicenow.snow\_record:

    username: "{{ snow.username}}"

    password: "{{ snow.passord}}"

    instance: "{{ snow.instance }}"

    state: present

    number: "{{snow\_incident\_result['record']['number']}}"

    data:

      work\_notes : "{{ resource\_name }} has been {% if resource\_changed %}created/updated{% else %}checked (no changes){% endif %}."

CopyExplain

At the end of the playbook run, you can close the incident, which will permanently record whatever information you ship from your playbook in your ITSM tool.

Microsoft Teams

While we covered Slack as the primary example in this chapter, Ansible also supports several Microsoft 365 products, including Microsoft Teams, via the community.general.office\_365\_connector\_card module. Microsoft 365 Connector cards are very powerful, and their configuration and, by extension, the Ansible module can get quite complicated; so rather than cover them here, I would recommend the following links as a starting point:

* https://docs.ansible.com/ansible/latest/collections/community/general/office\_365\_connector\_card\_module.html
* https://learn.microsoft.com/en-us/microsoftteams/platform/task-modules-and-cards/what-are-cards
* https://adaptivecards.io/

As you can see from the preceding links, connector cards can be as simple or complicated as you want. However, configuring them is probably worth a chapter all by itself, so let’s move on.

Summary of third-party services

One of the key takeaways I hope you get from this book is that automation is great; it is not only a real time saver, but using tools such as the ones we covered in the previous chapter, [*Chapter 16*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/16), *Introducing Ansible AWX and Red Hat Ansible Automation Platform*, can enable people who are not sys-admins or developers to execute their playbooks from a friendly web interface. We will look at this further in the final section of the chapter, where I will cover some real-world examples of how Ansible has been implemented in organizations I have worked with.

The modules we have covered in this section allow you to take your automation to the next level by not only allowing you to record the results but also automatically doing some housekeeping during your playbook run and having it notify your users.

For example, you need to deploy a new configuration to your server. Your service desk has made a change request for you to take action on the work within your ServiceNow installation.

Your playbook could be written so that before the change is actioned, it uses the fetch module to copy the configuration file to your Ansible Controller. The playbook could then use the servicenow.servicenow.snow\_record module to attach a copy of the existing configuration file to the change request, proceed to make the changes, and then automatically update the change request with the results.

Before we look at some real-world examples, let’s take a look at how you can debug your playbooks as they are running.

The Ansible playbook debugger

Ansible has a debugger built in. Let’s look at how you can build this into your playbook by creating a simple playbook with an error. As we have just mentioned, we are going to write a playbook that uses the community.general.say module. The playbook itself looks like this:

- name: "A simple playbook with a mistake"

  hosts: "localhost"

  debugger: "on\_failed"

  vars:

    message: "The task has completed and all is well"

    voice: "Daniel"

  tasks:

    - name: "Say a message on your Ansible host"

      community.general.say:

        msg: "{{ massage }}"

        voice: "{{ voice }}"

CopyExplain

There are two things to point out: the first is the mistake. As you can see, we are defining a variable named message, but when I came to use it in the task, I made a typo and entered massage instead. Luckily, as I developed the playbook, I instructed Ansible to use the interactive debugger whenever a task fails by setting the debugger option to on\_failed.

Debugging the task

Let’s run the playbook and see what happens:

$ ansible-playbook playbook.yml

CopyExplain

The first problem is that we are not passing a host inventory file, so there will be warnings that only the localhost is available; this is fine, as we want to run the Say module only on our Ansible Controller anyway:

[WARNING]: No inventory was parsed, only implicit localhost is available

[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost does not match 'all'

CopyExplain

Next, Ansible runs the play itself; this should result in a fatal error:

PLAY [A simple playbook with a mistake] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [Say a message on your Ansible host] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [localhost]: FAILED! => {"msg": "The task includes an option with an undefined variable. The error was: 'massage' is undefined. 'massage' is undefined\n\nThe error appears to be in '/Users/russ.mckendrick/Code/Learn-Ansible-Second-Edition/Chapter17/debugger/playbook.yml': line 12, column 7, but may\nbe elsewhere in the file depending on the exact syntax problem.\n\nThe offending line appears to be:\n\n  tasks:\n    - name: \"Say a message on your Ansible host\"\n      ^ here\n"}

CopyExplain

Typically, the playbook run will stop, and you will be returned to your shell; however, because we have instructed Ansible to drop into the interactive debugger, we now see the following prompt:

[localhost] TASK: Say a message on your Ansible host (debug)>

CopyExplain

From here, we can start to investigate the problem a little more; for example, we can review the error by typing the following command:

p result.\_result

CopyExplain

In Ansible, when using the debug module, the p command is used to prettify the output of a variable or expression. It stands for **pretty** or **pretty-print**. When you use p result.\_result in an Ansible debug task, it will display the value of result.\_result in a more readable and formatted way. The p command uses the pprint (**pretty-print**) function from the Python standard library to format the output.

Once you hit the *Enter* key, the results of the failed task will be returned:

{'\_ansible\_no\_log': False,

'failed': True,

'msg': 'The task includes an option with an undefined variable. The error was: \'massage\' is undefined. \'massage\' is undefined\n\nThe error appears to be in \'/Users/russ.mckendrick/Code/Learn-Ansible-Second-Edition/Chapter17/debugger/playbook.yml\': line 12, column 7, but may\nbe elsewhere in the file depending on the exact syntax problem.\n\nThe offending line appears to be:\n\n  tasks:\n    - name: "Say a message on your Ansible host"\n      ^ here\n'}

CopyExplain

Let’s take a closer look at the variables used in the task by typing the following:

p task.args

CopyExplain

This will return the two arguments we are using in the task:

{'msg': '{{ massage }}', 'voice': '{{ voice }}'}

CopyExplain

Now, let’s look at the variables that are available to the task using the following:

p task\_vars

CopyExplain

You may have noted that we instructed Ansible to execute the setup module as part of the playbook run, so the list of variables available to the task is very long:

          'inventory\_hostname': 'localhost',

          'inventory\_hostname\_short': 'localhost',

          'message': 'The task has completed and all is well',

          'module\_setup': True,

          'omit': '\_\_omit\_place\_holder\_\_7da4853be448a08d857e98fbabe7afe1b7c97d00',

          'play\_hosts': ['localhost'],

          'playbook\_dir': '/Users/russ.mckendrick/Code/Learn-Ansible-Second-Edition/Chapter17/debugger',

          'voice': 'Daniel'},

CopyExplain

As you can see, there is much information about the environment in which our playbook is being executed. In the list of variables, you will notice that all the information gathered by the setup modules starts with ansible\_, and our two variables are listed at the bottom.

We can find out more about these two variables by running the following commands:

p task\_vars['message']

p task\_vars['voice']

CopyExplain

This will display the contents of the variable:

[localhost] TASK: Say a message on your Ansible host (debug)> p task\_vars['message']

'The task has completed and all is well'

[localhost] TASK: Say a message on your Ansible host (debug)> p task\_vars['voice']

'Daniel'

CopyExplain

We know we are passing a misspelled variable to the msg argument, so we will make some changes on the fly and continue the playbook run. To do this, we are going to run the following command:

task.args['msg'] = '{{ message }}'

CopyExplain

This will update the argument to use the correct variable; we can now rerun the task by issuing the following command:

redo

CopyExplain

This will immediately rerun the task with the correct argument and, with any luck, you should hear, “*The task has completed, and all is well*:”

changed: [localhost]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

localhost: ok=1  changed=1  unreachable=0  failed=0. skipped=0  rescued=0. ignored=0

CopyExplain

As you can see from the preceding output, because we only have a single task, the playbook is completed. If we had more, it would carry on from where it left off. You can now update your playbook with the correct spelling and proceed with the rest of your day. Additionally, if we wanted to, we could have typed either continue or quit to proceed or stop, respectively.

Summary of the Ansible debugger

The Ansible debugger is a handy option to enable when you are working on creating complex playbooks; for example, imagine that you have a playbook that takes about 20 minutes to run, but it throws an error somewhere toward the end, say, 18 minutes after you first run the playbook.

Having Ansible drop into the interactive debugger shell not only means you can see precisely what is and isn’t defined, but it also means you don’t have to blindly make changes to your playbook and then wait another 18 minutes to see whether those changes resolved the fatal error.

Some real-world examples

Before we finish the chapter and the book, I will give a few examples of how I have used and interacted with Ansible over the last few years.

Automating a complex deployment

In this example, an application was distributed across several dozen servers in a public cloud. Each application component was installed on at least three different hosts and required updates in a specific order.

The application developers collaborated with the operations team to streamline the deployment process to create an Ansible Playbook. The playbook automated the following steps for each component of the application:

1. Put the application into maintenance mode by connecting to the targeted hosts and executing a specific command.
2. Create snapshots of all the costs involved in the deployment, ensuring a rollback point if needed.
3. Initiate the deployment process by pulling the latest code from the designated GitHub repository and executing a series of commands to update the application.
4. Verify the deployment’s success by connecting to the application’s API and running a set of health checks on each targeted host.
5. If the deployment and health checks pass successfully, take the application out of maintenance mode and proceed to the next component. However, if any tests fail, halt the deployment immediately and execute commands to revert the hosts to the previously taken snapshots, ensuring a safe rollback.

Prior to implementing Ansible automation, the manual execution of these deployment steps took several hours, as the application and operations teams had to co-ordinate and follow the process meticulously. This manual approach made deployments challenging and prone to human errors.

By automating the deployment tasks using Ansible, the teams could focus on handling the exceptions that arose due to genuine issues rather than mistakes caused by manual execution. Before the automation was put in place, errors were common during almost every release, with many hosts and complex manual steps involved.

The introduction of Ansible automation significantly improved the deployment process, reducing the time required and minimizing the risk of human errors. The playbook ensured consistency, reliability, and repeatability across multiple deployments, enabling the teams to deploy the application components more frequently and with greater confidence.

This example demonstrates how Ansible can tackle complex deployment scenarios, streamline processes, and enhance collaboration between development and operations teams in a public cloud environment.

Combining Ansible and other tools

In this real-world scenario, we collaborated with a team that had invested significant effort in developing their infrastructure automation using Terraform. Their Terraform code successfully deployed the infrastructure and performed basic host bootstrapping using a simple cloud-init script.

However, as the application requirements grew more complex, it became evident that additional automation was needed to effectively manage the application on the provisioned hosts. Instead of replacing the existing Terraform code, we introduced Ansible to complement the infrastructure automation.

To integrate Ansible with the existing Terraform workflow, we utilized the community.general.terraform module. This module allowed us to execute the Terraform deployment directly from within an Ansible playbook.

By leveraging this integration, we took the output generated by the Terraform deployment and passed the relevant information back to Ansible. This enabled Ansible to gather detailed information about the provisioned hosts and perform the necessary application bootstrapping tasks.

The combination of Terraform and Ansible proved to be a powerful solution:

* Terraform handled the infrastructure provisioning, ensuring the required resources were created and configured correctly in the target environment.
* Ansible took over the application management, utilizing the host information provided by Terraform to configure and deploy the application components seamlessly.

This approach allowed the team to maintain their existing Terraform codebase while extending the automation capabilities with Ansible. The integration between the two tools provided a seamless workflow, enabling the team to manage both the infrastructure and the application more effectively without having to throw away the code that they already had.

The team achieved a more comprehensive and efficient automation solution by choosing the right tools for specific tasks and leveraging their strengths. Terraforms infrastructure-as-code capabilities, combined with Ansible’s application management and orchestration features, resulted in a robust and flexible automation pipeline.

Deploying Ansible AWX

As discussed in [*Chapter 16*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/16), *Introducing Ansible AWX and Red Hat Ansible Automation Platform*, Ansible AWX is a powerful tool that offers a wide range of features beyond the basics. In addition to the core functionalities, Ansible AWX provides capabilities such as surveys, integration with identity services such as Microsoft Entra, and **role-based access controls** (**RBACs**) that enable granular access management for projects and templates.

Surveys in Ansible AWX allow you to create interactive forms that gather input from users before running a playbook. This feature is particularly useful when you need to collect specific information or parameters from end-users without exposing them to the underlying playbook complexities.

Integration with identity services, such as Microsoft Entra, enables seamless authentication and authorization for Ansible AWX users. This integration allows you to leverage existing user accounts and access controls, simplifying user management and ensuring secure access to Ansible AWX resources.

RBAC in Ansible AWX provides a flexible and granular way to manage user permissions. With RBAC, you can define roles and associate them with specific projects, templates, and other resources. This allows you to control who can access and execute specific playbooks, ensuring that users have the appropriate level of access based on their responsibilities and expertise.

In the following examples, we’ll explore how Ansible AWX has been utilized in various organizations that I have worked with to streamline processes, automate tasks, and empower teams to perform their duties effectively while maintaining security and governance.

Provisioning virtual machines

In this scenario, the IT team needed to provide a self-service portal for developers to provision **virtual machines** (**VMs**) across different environments, such as development, staging, and production. Each environment had specific requirements and configurations.

To streamline the process, Ansible AWX was deployed, and a survey was created to capture the necessary information from the developers. The survey included fields for specifying the desired operating system, VM size, environment, and other relevant parameters.

Upon submitting the survey, Ansible AWX triggered a playbook that automated the provisioning process. Based on the survey responses, the playbook dynamically generated the appropriate VM configurations and provisioned the VMs in the specified environment.

Additionally, the playbook integrated with the organization’s ticketing system, automatically creating a ticket with the VM details and linking it to the change management process for tracking and auditing purposes.

By leveraging Ansible AWX and surveys, the IT team empowered developers to provision VMs on-demand while maintaining control and governance over the process.

Managing application deployments

In another use case, a software development team needed to deploy their application across multiple environments, including development, QA, and production. Each environment had its own set of configurations and dependencies.

To simplify the deployment process, Ansible AWX was utilized. A survey was created to capture the necessary deployment parameters, such as the application version, target environment, and any specific configuration options.

The survey responses were then passed as variables to an Ansible playbook that was responsible for executing the deployment. The playbook handled the entire deployment process, including the following:

* Retrieving the specified application version from the artifact repository
* Configuring the target environment based on the provided parameters
* Deploying the application components and dependencies
* Running post-deployment tests and health checks
* Updating the deployment status in the organization’s project management tool

By using Ansible AWX and surveys, the development team could initiate deployments through a user-friendly interface, ensuring consistency and reducing the risk of manual errors. The playbook automated the complex deployment steps, saving time and effort for the team who needed the deployment while freeing up the time of the team who would have done the deployment.

Updating DNS records

In this example, the organization managed multiple **DNS** (or, to give it its full name, **domain name system**) zones across different providers, and they needed to allow front-line support teams to update DNS records without granting them direct access to the providers’ management consoles.

To achieve this, Ansible AWX was used. A survey was created to capture the necessary information for updating DNS records. The survey included fields specifying the domain name, record type (e.g., A, CNAME, MX), record value, and **time to live** (**TTL**).

Upon submitting the survey, Ansible AWX triggered a playbook that automated the DNS record update process. The playbook performed the following steps:

1. Validated the provided survey inputs to ensure data integrity and prevent invalid entries
2. Determined the appropriate DNS provider based on the domain name specified in the survey
3. Connected to the DNS provider’s API using the necessary credentials securely stored in Ansible Vault
4. Retrieved the existing DNS records for the specified domain and record type
5. Updated the DNS record with the new value and TTL provided in the survey
6. Saved the updated DNS record using the provider’s API
7. Logged the change in the organization’s change management system, such as ServiceNow, for tracking and auditing purposes

By using Ansible AWX, the front-line support teams could easily update DNS records without requiring direct access to the DNS providers’ management consoles. The playbook automated the complex steps involved in updating DNS records across multiple providers, ensuring consistency and reducing the risk of errors.

Additionally, the integration with the change management system provided a centralized record of all DNS changes, enabling easy tracking, auditing, and compliance with the organization’s change control processes.

These examples demonstrate how Ansible AWX can be leveraged to run tasks and simplify processes for end-users across different domains, such as infrastructure provisioning and application deployment. By combining Ansible AWX with surveys and integrating with existing tools and processes, organizations can enable self-service capabilities while maintaining control and governance over critical operations.

Summary

We have reached the end of the chapter and our book. I have been trying to think of a way to summarize Ansible; I believe the summary from the first edition of *Learn Ansible* still stands.

In response to a technical recruiter who reached out to him with a job role that required at least three years of Ansible experience when the tool had only been available for a short time, Ansible creator Michael DeHaan said the following in a now-deleted Tweet:

*“Anyone using Ansible for a few months is as good as anyone using Ansible for three years. It’s a simple tool on purpose.”*

That perfectly sums up my experience of Ansible and hopefully yours.

Once you know the basics, it is straightforward to move on and start building more complex playbooks quickly. These playbooks can assist with deploying basic code and applications as well as complex cloud and even physical architectures.

Reusing your roles and accessing an extensive collection of community-contributed roles and modules via Ansible Galaxy means you have many examples or quick starting points for your next project. So, you can roll your sleeves up and get stuck in a lot sooner than you would with other tools. Additionally, if Ansible cannot do something, the odds are that there is a tool it can integrate with to provide the missing functionality.

Going back to what we discussed back in [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*, being able to define your infrastructure and deployment in code in a repeatable and shareable way that encourages others to contribute to your playbooks should be the aim of starting to introduce Ansible into your day-to-day workflows.

Through this book, I hope you have begun to think of day-to-day tasks where Ansible could help you and save you time, and I wish you luck with developing your own playbooks.