KIT304 Server Administration and Security Assurance

Tutorial 15

Goal

In this tutorial you will learn about basic configuration management using the Ansible CM tool from RedHat.

Introduction

Configuration Management (often abbreviated to CM) is the practice of using tools to automate the management of system configurations. The technique has become increasingly valuable as a way to manage the ever-growing collections of servers, often cloud-based, that are required to provide the many applications that a typical modern business uses.

Today you'll get to know the basics of the Ansible CM tool. Ansible is relatively young, having been first released in 2012. It was acquired by RedHat (the owners of CentOS) in 2015, and can be used to manage systems of almost any type – be they Linux-based, Windows, or macOS.

Ansible is an agentless CM, meaning you don't need to install any software on the managed (client) systems. You do, however, need to configure the clients to accept password-less incoming SSH connections from the Ansible Server so that it can gather information about their current configuration (to work out what might need to be updated), and then actually carry out the necessary changes to the client to configure it to the desired state. That could mean creating a new user account, installing an application, creating a virtual web host, or any number of other things that a system administrator might need to do as part of their duties.

You'll start by configuring several clients so that they accept incoming SSH connections from the Ansible Server, and use Ansible in the most basic way by running simple commands from the command line. Then, you'll learn about Ansible Playbooks – a higher level approach that lets you write a document that describes a desired state for a client, and how to apply playbooks to clients. Finally, you'll learn a little about Ansible Roles – a way to structure larger Ansible projects into more easily managed component parts that can be re-used across playbooks.

Activity

Today you'll be setting up the CentOS 8 virtual machine as an Ansible server, and you will use it to manage the configuration of several other client systems, each of which will also be CentOS 8 instances. Ansible has been pre-installed on the CentOS VM but before you can use it, you'll need to do some configuration so that Ansible can communicate with the client machines that it will manage.

1. Start up the CentOS8 VM, log in as the **student** user, give the machine an IP address in the **`192.168.1.`*`x`*** subnet, open a terminal window, and become the **root** user. For the rest of this tutorial, we'll refer to this machine as the **Ansible Server**.

2. In the VMWare Fusion Virtual Machine Library, you'll find two machines named **Client1** and **Client2**. These are also CentOS 8 installations, identical to the CentOS8 virtual machine that you've just configured.

   Start up these client machines, and on each of them, log in as the **student** user, give them IP address in the **`192.168.1.`*`x`*** subnet, and open a terminal window. Ensure that you can ping each of the client VMs from the main CentOS VM (the Ansible Server).

3. On the Ansible server, add entries to the **`/etc/hosts`** file so that you can refer to each of the clients via a name (e.g., **`client1`** and **`client2`**) rather than via IP address.

Ansible communicates with its clients over the SSH protocol. Fortunately, since your two client systems are configured identically to the CentOS8 VM, they already have a configured and running SSH server. You've used it multiple times in earlier tutorials, when for example you've connected from Kali to CentOS in the terminal. Thus, you don't need to set up the SSH server on the two clients today, but SSH is not necessarily installed or running on many Linux distributions, so depending on the operating system of the client, you may need to install and/or configure it.

While you don't need to configure SSH, you are going to create an **SSH keypair** which will allow the server to securely connect to the clients without the need to enter a password. You'll also create a local user on each client that has password-less **`sudo`** privilege so that it can run commands (sent by the Ansible server) as the **root** user. To do that, follow these steps:

4. On ***each*** of the client systems, in the terminal window, become the **root** user, add a new user named **ansible** and make that user a member of the **`wheel`** group with the following commands:

   ```
   useradd ansible
   usermod -aG wheel ansible
   ```

   Being in the **`wheel`** group means that the **ansible** user can run commands with administrative privilege, which is necessary to do standard admin tasks like creating users, change the web server configuration, and so on – in other words, to do all of the things that you might want to be able to configure through Ansible.

   Even though the **ansible** user is now in the **`wheel`** group, the system will still require that they provide their password when they run commands with **`sudo`**. This isn't an option for processes that run as part of an automation system (like Ansible) so you also need to configure **sudo** to allow the ansible account to run commands via **sudo** without providing a password. To do that, create the file **/etc/sudoers.d/ansible** on both clients, and populate it with the following:

```
ansible ALL=(ALL) NOPASSWD:ALL
```

You'll test that this password-less **sudo** works in the next step.

5. Next, give the **ansible** user on both client systems a password that is long and random so that it can't be guessed by brute-force techniques. Use the same password for each client. You'll need to remember this password for the duration of this tutorial, so be sure to save it somewhere convenient. If you can't think of a random enough password on your own, enter the command **uuidgen** in a CentOS terminal window – the values it produces each time you run it are suitable for use as a random password.

   Once you've added a password on each client's **ansible** account, ensure that you can log in via **ssh** as that user:

   ```
   ssh ansible@clientx
   ```

   Once you've verified that the login works, test that the **ansible** user can execute sudo commands without being prompted for a password:

   ```
   sudo pwd
   ```

   This should display the current working directory *without prompting for a password*. If you are asked for a password, check that you followed the last part of step 4 correctly. If you can't find the problem, be sure to ask your tutor for help.

   Once you're sure that you can connect over **ssh** as the **ansible** user, and that the account can run password-less **sudo** commands, log out again (with control-D). Repeat this check for the **ansible** account on the other client.

6. The next step is to generate an SSH keypair that will allow the Ansible Server to connect securely over SSH to its clients without using a password at all. To do that, on the server in a terminal window running as the **root** user, enter the following command:

   ```
   ssh-keygen
   ```

   You'll be asked to specify a file save location, which defaults to **/root/.ssh/id_rsa**. Rather than use the default, enter **/root/.ssh/ansible-control**, to better reflect its purpose.

Next, you'll be asked to enter a passphrase for the private key. Passphrases protect keys in the event that they're stolen, but they require that someone provide the passphrase before a key can be used, and that's not convenient in automation scenarios, so just press enter when prompted for the passphrase.

The system will then generate a keypair, storing the private key in **`/root/.ssh/ansible-control`**, and the matching public key in **`/root/.ssh/ansible-control.pub`** – confirm this by looking in root's **`.ssh`** directory.

When using keypairs with SSH, you copy the public key to another system (or systems) that you want to connect to ahead of your first use, and keep the private key to yourself. When you want to log in to that remote system, your private key is used for the encryption. The remote system uses the matching public key that you provided earlier to decrypt the connection, and trusts that you've kept the private key private. Providing you've done so, it guarantees the CIA security requirements – encrypted communication that is safe from eavesdropping and modification, and where you can authenticate (and therefore trust) the origin.

To install your public key on each of your client machines, follow these steps:

7. Copy the public key from the Ansible Server to the **ansible** account on each of the clients with the following command:

   ```
   ssh-copy-id -i .ssh/ansible-control.pub ansible@clientx
   ```

   You'll be prompted to enter the password of the **ansible** account (the one you created back at step 5) as you run this for each client. This command actually appends the public key to the file **`authorized_keys`** in the **`.ssh`** directory of the **ansible** account on each of the clients.

   > ➢ The **`authorized_keys`** file is a standard **`ssh`** file that lists of all of the public keys whose matching private key will be accepted for password-less login to this account on this system.

8. With the public key installed in the **ansible** accounts of each of the clients, you can now check that password-less logins to each of the clients work. You do this with **`ssh`** by specifying the identify file (that is, the private key) that you want to use, with the **`-i`** option. For your two clients, that would be as follows:

   ```
   ssh -i .ssh/ansible-control ansible@clientx
   ```

   In both cases, if the login works you should see the `[ansible@localhost ~]$` shell prompt from the remote system without being prompted to enter a password. For each connection, be sure to log out with control-D. If you can't get password-less logins to work, ask your tutor for help.

Basic Ansible Configuration and Ad-hoc Commands

At this point, you've configured both clients such that they will accept password-less logins to the **ansible** account from the Ansible Server, and furthermore you've allowed that account to have the ability to run commands via **sudo** also without requiring a password. You're now ready to start doing some work with Ansible itself.

> ➢ As you may guess, the **`ansible`** account is now the equivalent of a **root** account, and can do anything the Ansible Server asks. One reason you've used a long, secure password on this account is to protect against brute-force password attacks. Production systems are often configured to block password logins over **`ssh`** on accounts like this, so that they can only be accessed via private key.

9. The details of which systems an Ansible server will manage are stored in an **inventory file**. The default inventory file which is **`/etc/ansible/hosts`** but you can create per-project inventory files if you need more flexibility. As root, in the terminal, open **`/etc/ansible/hosts`** in an editor, and check some of the examples in the comments, then move to the bottom of the file, and add the following details, substituting appropriate values for your client hostnames:

```
[clients]
client1
client2

[clients:vars]
ansible_user=ansible
ansible_ssh_private_key_file=~/.ssh/ansible-control
```

This sets up a group of hosts called **`clients`**, and defines two variables to be used when connecting to them – the account name on the remote system, and the SSH private key file to use.

You can have any number of collections of hosts under different section headers. For example, you could have **`[Launceston]`** and **`[Hobart]`** sections, so you could manage different computers in each city independently. The section that defines variables specific to a group of hosts always appears in an inventory file as **`[`*`section`*`:vars]`**. This allows you to have separate ansible usernames for different systems, or different private keys for different groups of systems. There are other ways you can define and use variables in Ansible, and you'll see some of them later in this tutorial.

> ➢ See https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html for more details on Ansible inventory files.

10. With your inventory file in place, you can now test that Ansible can connect to each of your client systems. The general format for ansible commands is

    ```
    ansible -m module -a 'moduleParameters' hostSet
    ```

    Not all modules require parameters. For example, try this simple Ansible command:

    ```
    ansible -m ping all
    ```

    This runs the **ping** module on all of the systems in the inventory file. Instead of **all**, you could provide the name of a group of systems in the inventory file (there is a group named **clients** that you could try) or you could list an individual system (e.g., *client1*).

    Assuming you haven't made any errors, you should get a SUCCESS response from each of the systems that you've run the **ping** module on.

11. Another module to try is the **shell** module. This lets you run arbitrary shell commands on your managed hosts. The general format is:

    ```
    ansible -m shell -a 'shellCommand' all
    ```

    For example:

    ```
    ansible -m shell -a 'groups ansible' all
    ansible -m shell -a 'uptime' all
    ```

12. The commands above run as the **ansible** user on the client, but without privilege. If you want to run shell commands that need administrative privilege, use the **-b --become-method=sudo** option, as follows:

    ```
    ansible -m shell -a 'service httpd start' -b --become-method=sudo all
    ```

    (If you try to run such commands *without* the **-b** and **--become-method** parameters, they will time-out and fail). Note that this command starts the web server on each client, but it also displays this important warning:

    ```
    [WARNING]: Consider using the service module rather than running 'service'.
    ```

    Many commonly used administrative operations have been developed into Ansible modules. You can run the same command with the **service** module like this:

    ```
    ansible -m service -a 'name=httpd state=started' all
    ```

When you use the service module in this way, it automatically runs at the client with the **`sudo`** command, so you don't have to specify this on the command line.

Note that the parameters are specified differently than they are when running this via the **`shell`** module. You're specifying which service (**`name=httpd`**) and the desired state (**`state=started`**).

This specification of the desired state, rather than an action to take (like **`start`**), is important to the idea of idempotence as discussed in the week 9 lecture. The other states you can specify, for the **`httpd`** service, include **`reloaded`**, **`stopped`**, and **`restarted`**.

13. Now that you've started them, try to connect to either (or both) of the web servers on the two CentOS clients from the FireFox browser on the Ansible Server. Why doesn't this work?

    ____

Ansible Playbooks

The Ansible commands you've used up to this point are all examples of **ad-hoc commands**. They're useful for being able to run quick commands across your managed inventory, but they're just the tip of the iceberg in terms of what Ansible is capable of.

For properly managed scenarios, you normally code the desired state of a system in one or more Ansible **Playbooks**. From the Ansible documentation:

> *"Playbooks record and execute Ansible's configuration, deployment, and orchestration functions. They can describe a policy you want your remote systems to enforce, or a set of steps in a general IT process."*
>
> *"Ansible Playbooks offer a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications. If you need to execute a task with Ansible more than once, write a playbook and put it under source control. Then you can use the playbook to push out new configuration or confirm the configuration of remote systems."*

Let's get started with a simple playbook to configure the firewall on the clients to allow incoming connections for **http** and **https** traffic. You've already done this on the command line in tutorial 9 (and elsewhere), with these commands:

```
firewall-cmd --permanent --zone=public --add-service=http
firewall-cmd --permanent --zone=public --add-service=https
firewall-cmd --reload
```

You can put the same functionality into a playbook.

14. Create a file named **firewall-up.yml**, and insert the following content into it. Be careful with indentation in this file. You cannot use tabs for indentation, but must only use spaces. All but the first two lines are indented by at least two spaces. Every level of indentation is another two spaces deep.

```
---
- hosts: clients
  become: yes
  become_method: sudo

  tasks:
    - name: enable http on firewall
      firewalld:
        service: http
        zone: public
        permanent: yes
        state: enabled

    - name: enable https on firewall
      firewalld:
        service: https
        zone: public
        permanent: yes
        state: enabled

    - name: reload firewall
      command: firewall-cmd --reload
```

If you study the file, you can see the parallels with the command-line equivalents. Notice at the top of the file you declare which **hosts** the playbook applies to (from the inventory file), and that it needs to become an admin user (via **sudo**) in order to run the commands (since you can't control the firewall without admin privilege).

Following that are three tasks. Each task has a **name** (which can be anything you want, and which is displayed as the playbook is run), and then the name of a module (**firewalld**, and **command** in this case). Nested under each module name are the named parameters it requires – in the case of the **command** module, where there's only one parameter (the command itself), the parameter is listed immediately after it.

To execute this playbook, enter the following command:

```
ansible-playbook firewall-up.yml
```

You'll see Ansible show its progress as the tasks in the playbook are run on each client. The first thing it does is gather "facts" – this is information that the module uses to ascertain the current state of the client – mostly, you'll see a returned status for these of **ok:** `[`*`clientName`*`]`. Then, it executes each task in the Playbook, and displays the result of each of them. For these, you'll most likely see the response **`changed: [`*`clientName`*`]`** which means the original state has been changed to match the new (desired) state in the Playbook (in this case, reconfiguring the firewall to open access for **http** and **https**). Finally, Ansible will display a recap, summarising the results of all of the operations in the playbook.

15. Since your **http** and **https** ports are now open, you should now be able to access the web servers running on the clients in the Firefox browser on the Ansible server – check that this is the case. If you can't access the web sites, check your work, and consider asking your tutor for help.

16. Return to the terminal, and run the same playbook again. On the first run, the first two tasks originally showed results of **`changed`**, but on the second run they're showing as **`ok`**. That's because they're already been run once (**http**/**https** access was opened) so no change needed to be made – that's the principle of idempotence in action.

    But there's one behaviour in the playbook that's not idempotent. It doesn't really matter in this scenario, but it would be better to fix it. Can you guess what it might be? (Hint: it's related to the third task).

    ____

17. To bring idempotency to the third task, you need to make some changes to the playbook. Do that with the following steps:

    - Open the playbook in your preferred editor.
    - Under the first task, add a new line consisting of **`register: http`** – this needs to be at the same indentation level as **`firewalld:`**, and immediately under the **`state: enabled`** entry. This tells Ansible to record (*register*) the state of the task in a variable called **`http`**. You can check that saved state later in the playbook.
    - Under the second task, add a new line consisting of **`register: https`** – this needs to be at the same indentation level as **`firewalld:`**, and immediately under the **`state: enabled`** entry.
    - Under the third task, add a new line consisting of **`when: http.changed or https.changed`** – this needs to be at the same indentation level as the **`command:`** line, and immediately under it. This tells Ansible to only run this task if either of the earlier tasks, whose output status is recorded in the variables **`http`** and **`https`**, have the value **`changed`**. In other words, only reload the firewall if the firewall ports were actually opened, but don't do anything if the ports were already open.

    Now re-run your playbook. This time, you should see that the reload task also doesn't happen, since the other two tasks didn't result in a change to the firewall.

18. Now make a copy of **`firewall-up.yml`** playbook, and name the copy **`firewall-down.yml`**. Open this in your preferred editor, and change the two **`state:`** lines, replacing the state **`enabled`** with **`disabled`**. Save your changes and exit the editor.

    You now have a new playbook that will close the **`http`**/**`https`** ports on the firewall. Run it (with **`ansible-playbook firewall-down.yml)`** and it should close down the open ports on the firewall. Check that this worked in Firefox – you should no longer be able to access the web servers running on the managed clients.

    Run this new playbook a second time – it should detect that the ports are already closed, and as a result will not reload the firewall.

So far, you've only experimented with a handful of built-in Ansible modules (**`ping`**, **`shell`**, **`service`**, **`firewalld`**, and **`command`**). Ansible has hundreds of built-in modules, and you can see a full list at https://docs.ansible.com/ansible/2.9/modules/list_of_all_modules.html for the version of Ansible that you're using on CentOS. In addition, you can write your own modules if the built-ins don't offer the functionality you need.

19. Take a look at the Ansible documentation for the **`user`** module (for creating user accounts on a system) at https://docs.ansible.com/ansible/2.9/modules/user_module.html. Use the information in this web page, and what you know from your existing playbooks, to write a new playbook named **`makedemoacct.yml`** that creates a regular user account named **`demo`** on all of hosts in the inventory. While the user module supports many different options, you should only need to use two parameters in your playbook – **`name`** and **`password`**.

    You can use the encrypted password for the **student** user (since you know what the actual password for this user is) which you can copy from the **`/etc/shadow`** file, or you can refer to the Ansible FAQ at https://docs.ansible.com/ansible/2.9/reference_appendices/faq.html to learn how to generate an encrypted password from a password and a salt.

    Test your playbook without actually executing it with the following command:

    ```
    ansible-playbook makedemoacct.yml --check
    ```

    This tests, as much as possible, that your playbook will run correctly, although it can't check parts that depend on the results of **`register:`** variables.

    Ensure that your playbook runs properly, and once created, that you can log in to both clients using the **`demo`** account. (If you need to run this more than once during your testing, on the clients, you can delete the **`demo`** account as the **`root`** user on the clients with **`userdel -r demo`**).

> ➢ A sample version of the **`makedemoacct.yml`** playbook is on the last page of this tutorial document, but try to develop your own version before resorting to using it. If your own version doesn't run, ask your tutor for assistance.

Ansible Roles

As the things you are trying to do in Ansible get more complex, it quickly gets impractical to put everything into a single playbook. To address this, you normally use **Ansible Roles** for larger projects. Roles consist of a series of YAML files organised into a predefined directory structure.

The main directories that are used in an Ansible Role are as follows:

`defaults`	Default values for variables (can be overridden)
`files`	Files that need to be copied to the managed systems by this role
`handlers`	Tasks that are run in response to a **`notify`** directive from another task.
`meta`	Metadata information about the role (author, dependencies, etc.)
`tasks`	The steps that need to be executed by the role (like the tasks in a Playbook)
`templates`	File templates that will be modified by a task before being copied to a managed system (e.g., an Apache Virtual Host template that needs to be completed before being installed on a web server client)
`vars`	Variables used for the role (overrides any default variable values)

All of these directories are optional, so you don't need to create them if you don't require them. Inside most of these directories you create a file **`main.yml`** that stores the information relevant to that directory. Thus, any variables used for the role are stored in **`vars/main.yml`** or potentially **`defaults/main.yml`**.

In this next exercise, you'll create a small Ansible role that configures the Apache web server on clients to support virtual hosts, and create a simple virtual host that you can test. This is in many ways an Ansible version of the Virtual Hosts exercise that you did manually back in tutorial 9. The difference here is that you can apply this role to any number of hosts easily through Ansible.

20. Start by creating the following subdirectory structure in **root**'s home directory on the Ansible Server:

    ```
    roles/
       vhost/
             handlers/
             tasks/
             templates/
    ```

    `roles` is a general area where you could store multiple roles – each of which exists as a subdirectory inside it. In this case, you're creating a single role named **`vhost`**, which will require three of the directories that all Ansible roles can use (in this case, **handlers**, **tasks**, and **templates**).

21. Inside **roles/vhost/handlers**, create the file **main.yml** with the following content:

```
---
- name: reload Apache
  service:
    name: httpd
    state: reloaded
```

    This handler will get called by a task when you need to reload Apache (after making configuration changes).

22. Inside **roles/vhost/templates**, create the file **index.html**, with the following content:

```
Hi, welcome to {{ domain }}!
```

    The value **{{ domain }}** will be replaced by the content of the **domain** variable when this template file gets copied to the managed client.

    Also in **roles/vhost/templates** create the file **virtualhost.conf** with the following content:

```
<VirtualHost *:{{ http_port }}>
    ServerName www.{{ domain }}
    ServerAlias {{ domain }}
    DocumentRoot /var/www/{{ domain }}/html
    ErrorLog /var/www/{{ domain }}//error_log
    CustomLog /var/www/{{ domain }}//access_log combined
</VirtualHost>
```

    You should recognise this content – it's essentially what you add as a virtual host configuration every time you create a new Apache virtual host. In this case, it's a template, and Ansible will again replace the value **{{ domain }}** with the content of the **domain** variable when this file gets copied to the managed client.

23. Next, inside **roles/vhost/tasks**, create the file **main.yml** with the following content. This is a longer file, containing multiple tasks that do the equivalent of what you would normally do manually when setting up virtual hosts on Apache. You should be able to copy-and-paste this from this PDF into the remote host to save some typing (and reduce the chance of errors):

```
---
- name: update Apache config to support named virtual hosts
  lineinfile:
    path: /etc/httpd/conf/httpd.conf
    regexp: NameVirtualHost
    line: NameVirtualHost *:80

- name: update Apache config to include hosts from sites-enabled
  lineinfile:
    path: /etc/httpd/conf/httpd.conf
    regexp: sites-enabled
    line: IncludeOptional sites-enabled/*.conf

- name: create the directories we need
  file:
    path: "{{ item }}"
    state: directory
  with_items:
  - "/etc/httpd/sites-enabled"
  - "/var/www/{{ domain }}"
  - "/var/www/{{ domain }}/html"

- name: create the apache log files and set their security context
  file:
    path: "{{ item }}"
    state: touch
    seuser: system_u
    setype: httpd_log_t
  with_items:
  - "/var/www/{{ domain }}/error_log"
  - "/var/www/{{ domain }}/access_log"

- name: create virtualhost file from template
  template:
    src: "virtualhost.conf"
    dest: "/etc/httpd/sites-enabled/{{ domain }}.conf"
  notify: reload Apache

- name: create index.html from template
  template:
    src: "index.html"
    dest: "/var/www/{{ domain }}/html/index.html"
```

Study the file you've just created carefully. You can probably work out what almost all of the tasks do just from their names. The fourth task creates the **error_log** and **access_log** files you normally create with **touch**, and sets their SELinux security context appropriately at the same time – something you do manually with the **chcon** command. Notice too that the second last task issues a notification to reload Apache once the virtual host configuration file has been created. That causes the handler named **reload Apache** (in **roles/vhost/handlers/main.yml**) to be run.

24. Finally, in **root**'s home directory (or the same location where you created the **roles** directory), create the file **playbook.yml** with the following content:

```
---
- hosts: all
  become: true
  roles:
    - vhost
  vars:
    http_port: 80
    domain: example.com
```

Notice that this file defines two variables (**http_port** and **domain**) which are used as the values to be substituted in the relevant templates when the role is executed. Most importantly in this file is the section named **roles**, which declares that all hosts that this playbook references must have the **vhost** role applied to them. If your hosts also required a **mysql** server to be installed, you would create a separate similar directory tree to the **vhost** tree named **mysql** (inside the **roles** directory), and then you would list that new role to the same **roles** section of this playbook file.

25. To apply your new role to your hosts, do the following:

- First, add an entry to **/etc/hosts** on the Ansible Server that points **example.com** to one of your managed clients.

- Next, check that your role and playbook files are all in order by running **ansible-playbook** with the **--check** option:

```
ansible-playbook playbook.yml --check
```

- If everything checks out OK, run the playbook with:

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

Assuming there were no errors, you should now be able to open the website **example.com** in the Firefox browser on the Ansible Server (you will need to rerun your earlier **firewall-up** playbook if your firewalls are still closed from step 18). Assuming your firewall is open, if you can't load the web site be sure to ask your tutor for assistance.

Conclusion

You have now reached the end of the Web, Database and Containers module. During this module you have learned how to set up web and database servers on both CentOS and Windows Servers, and in Docker containers, and you've learned the basics of Ansible – a widely-used configuration management tool. You have also gained some experience with DNS and TLS configuration. Don't forget that your next tutorial will be the 3rd practical exam. Spend any remaining time you have at the end of this session going over previous material that you haven't yet completed, practice your web and database skills, and be sure to make use of your tutor in tutorials.

Skills to Master

To successfully complete this tutorial, you must have mastered the following skills:

- create SSH public/private keypairs, and install the public key on clients, to allow password-less logins via the matching private key
- configure **sudo** to allow password-less use for a user
- configure and use Ansible sufficiently to be able to use ad-hoc commands on managed clients
- create Ansible playbooks to perform simple operations such as opening a firewall, or adding a user to a managed client
- create simple Ansible roles, for performing more sophisticated operations on managed clients

You could be required to demonstrate any of these skills in this module's practical exam.

Sample Playbooks

```
# create a user with a specific password
---
- hosts: clients
  become: yes
  become_method: sudo

  tasks:
    - name: create a demo user
      user:
        name: demo
        password: $6$T5Tjn$tFAJV/UciGEd7qMItGZG7Z3JswVWGMgie4Qsnu.EY04WBFF3M… (truncated)
```