Mastering Ansible
(From Beginner to Pro in 15 Minutes)

By : Ram Nath (Shri Ram Nath Om Parkash Bamal' | LinkedIn)

What is Ansible?

Ansible is an open-source automation tool, or platform, used for IT tasks such as configuration management, application deployment, intraservice orchestration, and provisioning. It aims to provide a simple, yet powerful way to automate and manage your IT infrastructure.

Benefits of Ansible

Open Source: Ansible is available for free and has a vibrant community for support.
Simplicity: Requires no special coding skills due to its use of YAML for playbook writing.
Power and Flexibility: Capable of modeling complex IT workflows; customizable for various configurations.
Agentless: No agents/software need to be installed on the client systems you are automating, reducing overhead.
Efficient: Since there's no extra software to install on nodes, more resources are available for your applications.
SSH-Based: Utilizes SSH for secure network communication, simplifying password-less authentication with managed nodes.

What Ansible Can Do

Configuration Management: Automates the process of managing configuration changes to ensure systems are maintained in a desired, consistent state.
Orchestration: Coordinates complex multi-tier deployments and workflows to ensure applications and services are deployed across various environments in sync.
Application Deployment: Streamlines the deployment of applications, ensuring they are correctly configured across environments.
Provisioning: Sets up various servers, cloud instances, and other infrastructure components.
Security and Compliance: Enforces security policies and compliance standards across your IT infrastructure.

Architecture of Ansible

Management Node: The controlling node where Ansible is installed and where playbooks are run from.
Host Nodes: The managed nodes that are controlled by the management node.
Host Inventory: A list of managed nodes. This list is stored in a simple file which Ansible references to identify the systems it can communicate with.
Playbooks: YAML files that describe the desired states of your systems, the tasks to be executed, and the order in which they should be executed.
Ansible.cfg: Configuration file for Ansible, specifying settings such as the inventory file location and default user settings.

Ansible Modules

Ansible modules are the building blocks for Ansible playbooks. They are essentially standalone scripts that can be used by the Ansible API, or by the **ansible** or **ansible-playbook** command-line tools, to perform automation tasks on remote hosts. Modules can do things like installing software, copying files, managing services, working with various cloud platforms, and much more. When you run a playbook, Ansible invokes the specified modules, with the specified arguments, on the targeted hosts.

Key Points About Ansible Modules:

- **Granularity:** Modules are designed to enable a specific task or a collection of related tasks, such as managing packages with the **apt** module or handling files with the **file** module.
- **Idempotency:** A core principle in most Ansible modules is idempotency, meaning that running the same module multiple times in a row with the same parameters will result in the same state of the system without performing unnecessary operations.
- **Wide Variety:** There are hundreds of modules included with Ansible, catering to various tasks and operations across different platforms, including Linux/Unix, Windows, cloud services, networking devices, and more.
- **Custom Modules:** If the standard modules don't cover your needs, you can write your own custom modules. Ansible modules can be written in any language that can return JSON, but Python is the most common choice.

- **Categories:** Modules are categorized based on their functionality. Some categories include system, network, cloud, monitoring, packaging, and database modules.

Example Usage:
In an Ansible playbook, you define tasks that call specific modules and pass them the necessary arguments. Here's a simple example:

```
- name: Ensure nginx is installed
  ansible.builtin.yum:
    name: nginx
    state: present
```

In this example, the **ansible.builtin.yum** module is used to ensure that the **nginx** package is installed. The **name** argument specifies the package, and the **state** argument specifies that the package should be **present** on the system.

Finding Modules:

You can find the documentation for all Ansible modules on the Ansible Documentation website. The **ansible-doc** command-line tool is also a handy way to view documentation for installed modules directly from your terminal.

Modules are a powerful aspect of Ansible's automation capabilities, providing a wide range of options for managing your infrastructure efficiently and effectively.

Ansible Tower

Ansible Tower is the enterprise offering from Ansible that provides a web-based user interface for managing playbooks. It adds advanced features like role-based access control, job scheduling, integrated notifications, and graphical inventory management. Tower makes it easier to scale automation, manage complex deployments, and improve process reliability.

Example: Deploying a Web Application with Ansible

Scenario: You need to deploy a web application to a set of web server nodes. This involves:

1. Ensuring the web servers are installed.
2. Configuring the web servers according to a specific configuration.
3. Deploying the application code from a version control system.

Steps:
1. Create an Inventory File: This file lists all the web server nodes.

--

[webservers]

server1.example.com

server2.example.com

2. Write a Playbook: The playbook describes the tasks to be performed on the web servers.

- name: Deploy Web Application

hosts: webservers

tasks:

- name: Install nginx

ansible.builtin.yum:

name: nginx

state: present

- name: Copy nginx configuration

ansible.builtin.copy:

src: /path/to/nginx.conf

dest: /etc/nginx/nginx.conf

- name: Copy application code

ansible.builtin.git:

repo: 'https://example.com/repo/app.git'

dest: /var/www/html/app

- name: Ensure nginx is running

ansible.builtin.service:

name: nginx

state: started

enabled: yes

3. Run the Playbook: Execute the playbook from your management node.

--

ansible-playbook deploy-app.yml

This example demonstrates how Ansible automates the process of deploying a web application to a set of servers with minimal manual intervention, illustrating its power and simplicity in automating IT infrastructure.

Advanced level Ansible Playbook for DevOps:

Creating a Docker image from .NET code hosted in a GitHub repository and pushing it to an Azure Container Registry (ACR) involves several steps, including cloning the repository, building the Docker image, logging into ACR, and pushing the image. Below is an Ansible playbook that outlines these steps.

Prerequisites:

- Ansible control node with Docker and the Azure CLI installed.
- Access to an Azure Container Registry.
- Dockerfile in the root of your .NET GitHub repository.

Playbook Explanation:

1. **Clone the GitHub Repository**: Downloads the .NET code from the specified GitHub repository.
2. **Build Docker Image**: Builds a Docker image from the Dockerfile found in the cloned repository.
3. **Login to Azure Container Registry**: Uses the Azure CLI to log into your Azure Container Registry.
4. **Tag Docker Image**: Tags the Docker image for the Azure Container Registry.
5. **Push Docker Image to ACR**: Pushes the tagged Docker image to your Azure Container Registry.

Ansible Playbook:

```
---
- name: Build and Push Docker Image to Azure Container Registry
  hosts: localhost
  vars:
    github_repo: "https://github.com/your-username/your-dotnet-app.git"
    project_name: "your-dotnet-app"
    image_name: "youracrname.azurecr.io/your-dotnet-app"
    image_tag: "latest"
    azure_acr_name: "youracrname"

  tasks:
    - name: Clone .NET code from GitHub repository
      ansible.builtin.git:
        repo: "{{ github_repo }}"
        dest: "./{{ project_name }}"
        clone: yes
        update: yes

    - name: Build Docker image from Dockerfile
      community.docker.docker_image:
        build:
          path: "./{{ project_name }}"
        name: "{{ image_name }}"
        tag: "{{ image_tag }}"
        source: build
```

```
- name: Log in to Azure Container Registry
  ansible.builtin.command:
    cmd: "az acr login --name {{ azure_acr_name }}"
  register: login_output
  failed_when: "'Login Succeeded' not in login_output.stdout"

- name: Tag Docker image for ACR
  community.docker.docker_image:
    name: "{{ image_name }}"
    repository: "{{ image_name }}"
    tag: "{{ image_tag }}"
    push: yes

- name: Push Docker image to Azure Container Registry
  community.docker.docker_image:
    name: "{{ image_name }}"
    tag: "{{ image_tag }}"
    push: yes
    source: local
```

Important Notes:

- Replace **your-username/your-dotnet-app.git**, **your-dotnet-app**, **youracrname.azurecr.io/your-dotnet-app**, **latest**, and **youracrname** with your actual GitHub repository URL, project name, image name (including ACR URL), image tag, and Azure ACR name.
- Ensure your Ansible control node is configured with the necessary permissions to access your Azure Container Registry and has the Azure CLI and Docker installed.

- This playbook uses the **community.docker.docker_image** module to manage Docker images, which requires the Docker SDK for Python. Ensure it is installed (**pip install docker**).

This playbook provides a basic framework for automating the process of building a Docker image from .NET code and pushing it to Azure ACR. You might need to adjust paths, names, and tags according to your specific project.