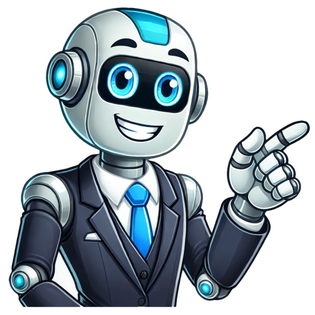
Continue



Ansible best practices

Blog Ansible community forum Documentation Ansible Community Documentation Ansible These concepts apply to all Ansible activities and artifacts. Whenever you can, do things simply. Use advanced features only when necessary, and select the feature that best matches your use case. For example, you will probably not need vars, vars_files, vars_prompt and --extra-vars all at once, while also using an external inventory file. If something feels complicated, it probably is. Take the time to look for a simpler solution. Keep your playbooks, roles, inventory, and variables files in git or another version control system and make commits with meaningful comments to the repository when you make changes. Version control gives you an audit trail describing when and why you changed the rules that automate your infrastructure. You can change the output from Ansible CLI commands using Callback plugins. To ensure that your automation project is easy to understand, modify, and share with others, you should avoid configuration-dependent content. For example, rather than referencing an ansible.cfg as the root of a project, you can use magic variables such as playbook_dir or role_name to determine paths relative to known locations within your project directory. This can help to keep automation content flexible, reusable, and easy to maintain. For more information, see special variables. These tips help make playbooks and roles easier to read, maintain, and debug. Generous use of whitespace, for example, a blank line before each block or task, makes a playbook easy to scan. Play, task, and block - name:’s are optional, but extremely useful. In its output, Ansible shows you the name of each named entity it runs. Choose names that describe what each play, task, and block does and why. For many modules, the state parameter is optional. Different modules have different default settings for state, and some modules support several state settings. Explicitly setting state: present or state: absent makes playbooks and roles clearer. Even with task names and explicit states, sometimes a part of a playbook or role (or inventory/variable file) needs more explanation. Adding a comment (any line starting with #) helps others (and possibly yourself in the future) understand what a play or task (or variable setting) does, how it does it, and why. Use fully qualified collection names (FQCN) to avoid ambiguity in which collection to search for the correct module or plugin for each task. For builtin modules and plugins, use the ansible.builtin collection name as a prefix, for example, ansible.builtin.copy. These tips help keep your inventory well organized. With cloud providers and other systems that maintain canonical lists of your infrastructure, use dynamic inventory to retrieve those lists instead of manually updating static inventory files. With cloud resources, you can use tags to differentiate production and staging environments. A system can be in multiple groups. See How to build your inventory and Patterns: targeting hosts and groups. If you create groups named for the function of the nodes in the group, for example, webservers or dbservers, your playbooks can target machines based on function. You can assign function-specific variables using the group variable system, and design Ansible roles to handle function-specific use cases. See Roles. You can keep your production environment separate from development, test, and staging environments by using separate inventory files or directories for each environment. This way you pick with -i what you are targeting. Keeping all your environments in one file can lead to surprises! For example, all vault passwords used in an inventory need to be available when using that inventory. If an inventory contains both production and development environments, developers using that inventory would be able to access production secrets. You should encrypt sensitive or secret variables with Ansible Vault. However, encrypting the variable names as well as the variable values makes it hard to find the source of the values. To circumvent this, you can encrypt the variables individually using ansible-vault encrypt_string, or add the following layer of indirection to keep the names of your variables accessible (by grep, for example) without exposing any secrets: Create a group_vars/ subdirectory named after the group. Inside this subdirectory, create two files named vars and vault. In the vars file, define all of the variables needed, including any sensitive ones. Copy all of the sensitive variables over to the vault file and prefix these variables with vault_. Adjust the variables in the vars file to point to the matching vault_ variables using jinja2 syntax: db_password: "{{ vault_db_password }}". Encrypt the vault file to protect its contents. Use the variable name from the vars file in your playbooks. When running a playbook, Ansible finds the variables in the unencrypted file, which pulls the sensitive variable values from the encrypted file. There is no limit to the number of variable and vault files or their names. Note that using this strategy in your inventory still requires all vault passwords to be available (for example for ansible-playbook or AWX/Ansible Tower) when run with that inventory. These tips apply to using Ansible, rather than to Ansible artifacts. Reduce complexity with portable container images known as Execution Environments. Testing changes in a staging environment before rolling them out in production is always a great idea. Your environments need not be the same size, and you can use group variables to control the differences between environments. You can also check for any syntax errors in the staging environment using the flag --syntax-check such as in the following example: ansible-playbook --syntax-check Use the serial keyword to control how many machines you update at once in the batch. See Controlling where tasks run: delegation and local actions. Group variables files and the group_by module work together to help Ansible execute across a range of operating systems and distributions that require different settings, packages, and tools. The group_by module creates a dynamic group of hosts that match certain criteria. This group does not need to be defined in the inventory file. This approach lets you execute different tasks on different operating systems or distributions. For example, the following play categorizes all systems into dynamic groups based on the operating system name: - name: Talk to all hosts just so we can learn about them hosts: all tasks: - name: Classify hosts depending on their OS distribution ansible.builtin.group_by: key: os_{{ ansible_facts['distribution'] }} Subsequent plays can use these groups as patterns on the hosts line as follows: - hosts: os_CentOS gather_facts: False tasks: # Tasks for CentOS hosts only go in this play. - name: Ping my CentOS hosts ansible.builtin.ping: You can also add group-specific settings in group vars files. In the following example, CentOS machines get the value of ‘42’ for asdf but other machines get ‘10’. You can also use group vars files to apply roles to systems as well as set variables. --- # file: group_vars/all asdf: 10 --- # file: group_vars/os_CentOS.yml asdf: 42 Note All three names must match: the name created by the group_by task, the name of the pattern in subsequent plays, and the name of the group vars file. You can use the same setup with include_vars when you only need OS-specific variables, not tasks: - name: Use include_vars to include OS-specific variables and print them hosts: all tasks: - name: Set OS distribution dependent variables ansible.builtin.include_vars: "os_{{ ansible_facts['distribution'] }}.yml" - name: Print the variable ansible.builtin.debug: var: asdf This pulls in variables from the group_vars/os_CentOS.yml file. Are you tired of manually configuring servers, deploying applications, and managing infrastructure? Look no further than Ansible, a powerful automation tool that can simplify your work and streamline your workflow. In this article, we'll share the best practices, tips, and tricks you need to know to use Ansible effectively and efficiently. What is Ansible? Before diving into the best practices, let's briefly review what Ansible is and how it works. Ansible is an open-source automation tool that allows you to automate IT tasks such as provisioning infrastructure, configuration management, application deployment, and security policies. Ansible uses a YAML-based language to define a set of instructions called playbooks, which describe the desired state of your infrastructure. Playbooks can be executed from the command line or integrated with other tools such as Jenkins, GitLab, or Tower. Best Practices Without further ado, let's explore the best practices for using Ansible effectively. 1. Organize Your Playbooks One of the most important aspects of using Ansible is organizing your playbooks in a logical and consistent manner. A well-organized playbook structure can help you understand the infrastructure's configuration, debug issues, and improve readability. Here are some tips to follow: Use a consistent naming convention for your playbook files and roles. For example, use webserver.yml for a playbook that deploys a web server role. Break down your playbook into small, reusable roles that can be shared across multiple playbooks. Define variables in separate files or in inventory groups to make it easier to manage and reuse them. Use comments and documentation to explain what each task and role is doing and why. 2. Use Ansible Galaxy for Reusable Roles Ansible Galaxy is a repository of pre-written roles that you can use to automate your infrastructure. Galaxy roles cover a wide range of use cases, from deploying applications to configuring security settings. By using Ansible Galaxy, you can save time and avoid reinventing the wheel. Here are some tips to use Ansible Galaxy effectively: Use role dependencies to automatically download and install the required roles. Use role tags to selectively execute specific roles in a playbook. Use role variables to customize the behavior of a role. Contribute to the community by submitting your own roles to Ansible Galaxy. 3. Use Ansible Vault for Sensitive Data Ansible Vault is a built-in feature that allows you to encrypt sensitive data such as passwords, API keys, and SSH keys. Ansible Vault uses AES256 encryption to protect your data, and you can use it to encrypt variables, files, and even entire playbooks. Here are some tips to use Ansible Vault effectively: Use a strong encryption passphrase and keep it in a secure location. Use the ansible-vault command to encrypt and decrypt files and variables. Store encrypted files and variables in a separate directory, and add them to your .gitignore file. Use --vault-password-file option to specify the path to your vault password file. 4. Use Ansible's Idempotent Nature One of Ansible's most powerful features is its idempotent nature, which means that running the same playbook multiple times should result in the same consistent state. This allows you to safely run playbooks repeatedly without worrying about breaking the infrastructure's configuration. Here are some tips to use Ansible's idempotent nature effectively: Always test your playbooks on a small scale before running them in production. Avoid making destructive changes that could cause data loss or downtime, such as deleting a critical file or restarting a production server. Use the ignore_errors and failed_when options to handle error conditions gracefully. Use conditional statements such as when or changed_when to ensure that tasks are only executed when necessary. 5. Use Ansible Roles for Modularity Ansible Roles provide a modular way to organize your playbooks and make them more readable and maintainable. Roles are collections of tasks, files, templates, and variables that are grouped together, making it easy to reuse and share them across different playbooks. Here are some tips to use Ansible Roles effectively: Use a consistent role directory structure, with a tasks directory for defining the role's main tasks, a vars directory for defining role-specific variables, and a templates directory for templates. Use the include_role and import_role modules to include and reuse roles in your playbooks. Use the default and override directories to manage default and overridden role variables. Use role dependencies to automatically install required roles. 6. Use Ansible's Debugging and Testing Tools Ansible provides a wealth of debugging and testing tools to help you troubleshoot your playbooks and ensure that they work as intended. Here are some tips to use Ansible's debugging and testing tools effectively: Use the --check option to perform a dry-run of your playbook and check for syntax errors and configuration issues. Use the --syntax-check option to validate the syntax of your playbook. Use the debug module to display debug messages and variables during playbook execution. Use the assert module to test the results of a task against an expected condition. Conclusion By following these best practices, you can make the most of Ansible's powerful automation capabilities and simplify your IT tasks. Organizing your playbooks, using Ansible Galaxy for reusable roles, leveraging Ansible Vault for sensitive data, taking advantage of Ansible's idempotent nature, using Ansible Roles for modularity, and using Ansible's debugging and testing tools will help you streamline your workflow and reduce your workload. Happy automating!Editor Recommended Sites AI and Tech News Best Online AI Courses Classic Writing Analysis Tears of the Kingdom Roleplay Zero Trust Security - Cloud Zero Trust Best Practice & Zero Trust implementation Guide: Cloud Zero Trust security online courses, tutorials, guides, best practice Tactical Roleplaying Games - Best tactical roleplaying games & Games like mario rabbids, xcom, fft, ffbe wotv: Find more tactical roleplaying games like final fantasy tactics, wakfu, ffbe wotv Datascience News: Large language mode LLM and Machine Learning news LLM Model News: Large Language model news from across the internet. Learn the latest on llama, alpaca Developer Painpoints: Common issues when using a particular cloud tool, programming language or framework Here are some tips for making the most of Ansible and Ansible playbooks. You can find some example playbooks illustrating these best practices in our ansible-examples repository. (NOTE: These may not use all of the features in the latest release, but are still an excellent reference!). The following section shows one of many possible ways to organize playbook content. Your usage of Ansible should fit your needs, however, not ours, so feel free to modify this approach and organize as you see fit. One crucial way to organize your playbook content is Ansible’s “roles” organization feature, which is documented as part of the main playbooks page. You should take the time to read and understand the roles documentation which is available here: Roles. The top level of the directory would contain files and directories like so: production # inventory file for production servers staging # inventory file for staging environment group_vars/ group1.yml # here we assign variables to particular groups group2.yml host_vars/ hostname1.yml # here we assign variables to particular systems hostname2.yml library/ # if any custom modules, put them here (optional) module_utils/ # if any custom module_utils to support modules, put them here (optional) filter_plugins/ # if any custom filter plugins, put them here (optional) site.yml # master playbook webservers.yml # playbook for webserver tier dbservers.yml # playbook for dbserver tier roles/ common/ # this hierarchy represents a "role" tasks/ # main.yml #