**Process of Developer Code in to Galaxy Architecture**

**Steps to Handle External Resources (Custom Files/Binaries) in a Galaxy Architecture**

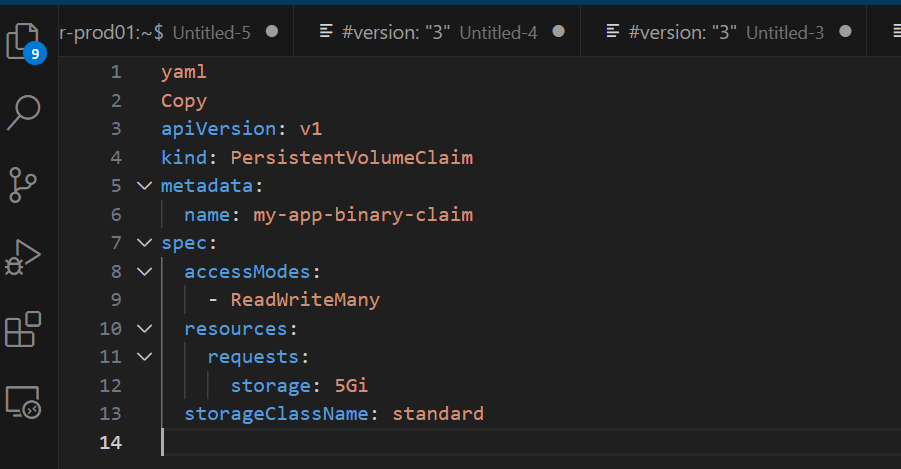
**Step 1: Identify and Classify External Resources**

* **Custom Files**: These could be configuration files, JSON files, YAML files, etc.
* **Binaries/Executables**: If the application depends on certain pre-compiled binaries or libraries that are not part of the standard container image.
* **Assets/Static Files**: Media, images, reports, or other assets that the application might use or serve.
* **Data Files**: External datasets or large files that the app interacts with (e.g., machine learning models, customer data, or logs).

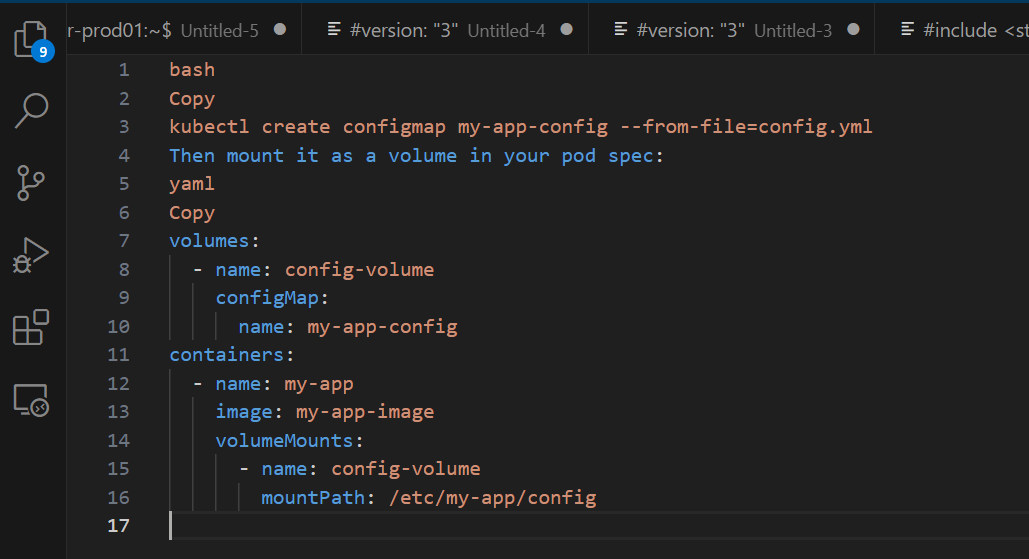
**Step 2: Storing and Accessing External Resources**

Depending on the nature of the external resource, you may choose one of the following methods to handle it:

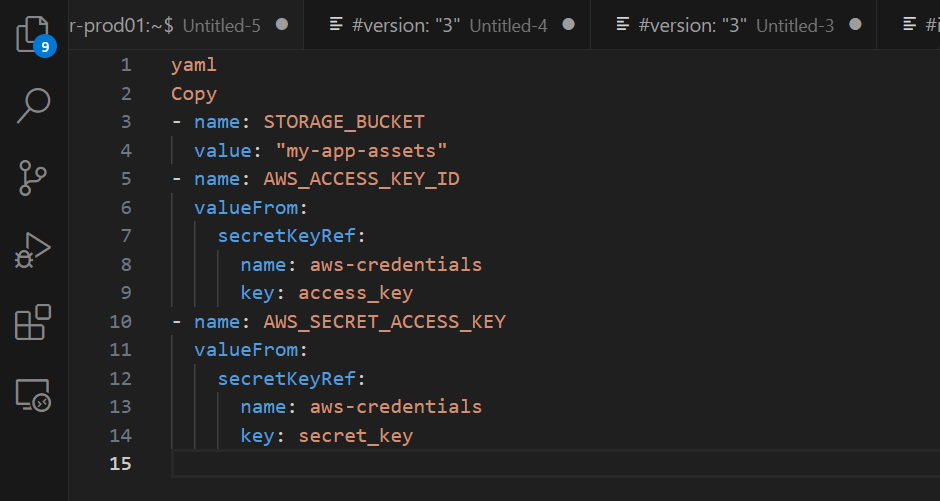
1. **Persistent Volumes (PVs) in OpenShift/Kubernetes**:
   * **For Custom Files and Binaries**:
     + If the files/binaries are required to be persistent and shared across pods (e.g., for application configuration or binaries), you can store them in **Persistent Volumes (PVs)** and **Persistent Volume Claims (PVCs)**.
     + OpenShift supports **StorageClass** to define various types of storage backends (e.g., NFS, Ceph, GlusterFS, AWS EBS, etc.).
     + Example:



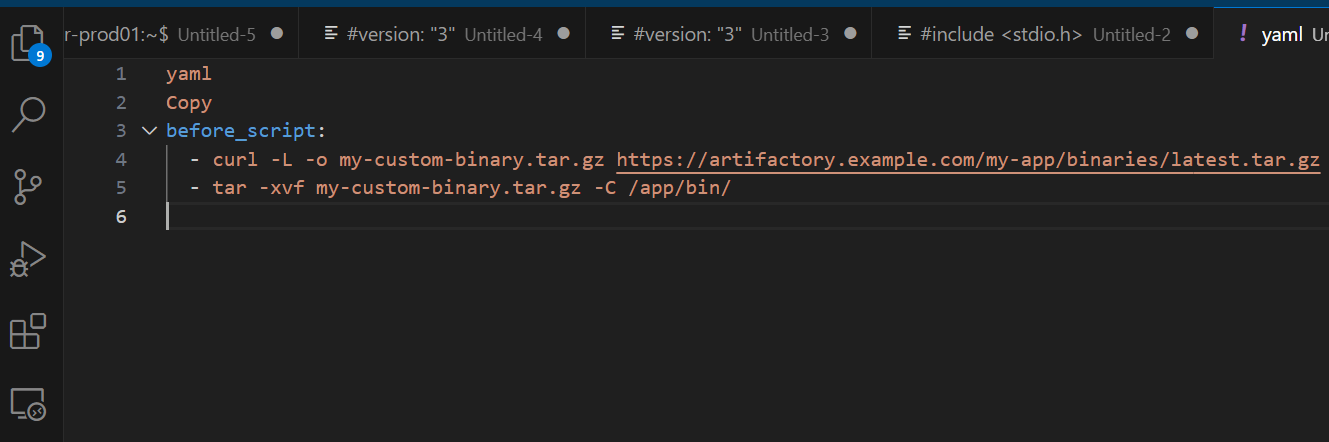
1. **ConfigMaps and Secrets**:
   * **For Configuration Files or Sensitive Data**:
     + **ConfigMaps** are great for storing non-sensitive configuration data, which the application can mount as files.
     + **Secrets** are used for sensitive data like passwords, API keys, certificates, etc.
     + Both **ConfigMaps** and **Secrets** can be easily mounted as volumes inside your OpenShift Pods or can be injected as environment variables.
     + Example: Store a configuration file using a ConfigMap:



1. **Object Storage** (Cloud Providers or Self-Managed):
   * **For Larger Binaries, Data Files, and Assets**:
     + If your files are too large or need to be accessed by multiple services, you may want to store them in a cloud-native object storage system, such as AWS S3, Google Cloud Storage, or OpenShift’s **MinIO** (if using an OpenShift-managed object storage solution).
     + You can store the binaries, static files, or datasets in the object storage and provide the application with access to them via appropriate credentials or tokens.
     + Integration with object storage can be done via environment variables or mounting as a volume.
     + Example:



1. **Artifact Repository** (e.g., Nexus, Artifactory, GitLab Registry):
   * **For Custom Binaries/Dependencies**:
     + If the app relies on custom binaries (e.g., pre-compiled libraries or tools), you can use an **artifact repository** like **Nexus**, **Artifactory**, or **GitLab’s container registry** to store the binaries.
     + During the CI/CD pipeline (in GitLab CI/CD), the build process will fetch the required binaries from the artifact repository and package them inside the container image.
     + Example: In the .gitlab-ci.yml file, you could add a step to pull binaries from the artifact repository:

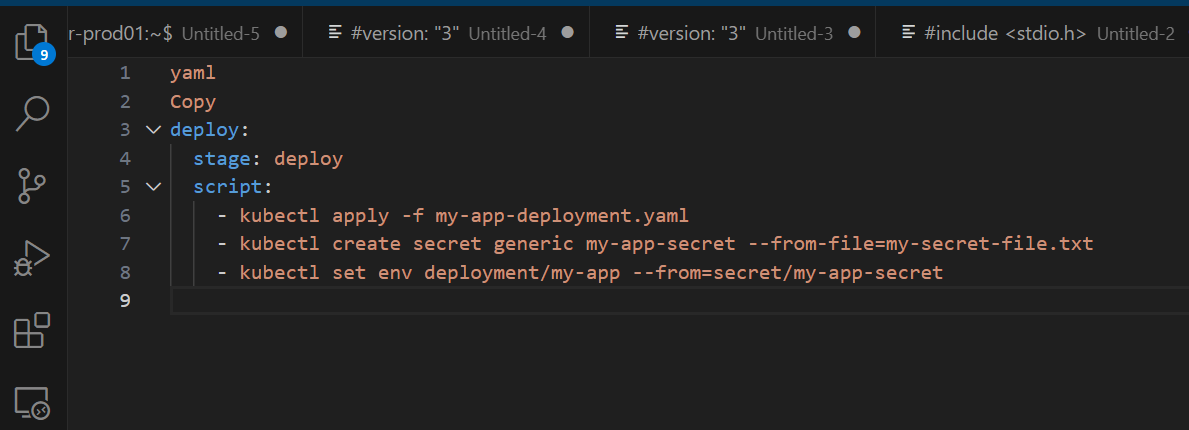


**Step 3: Managing Dependencies in the CI/CD Pipeline**

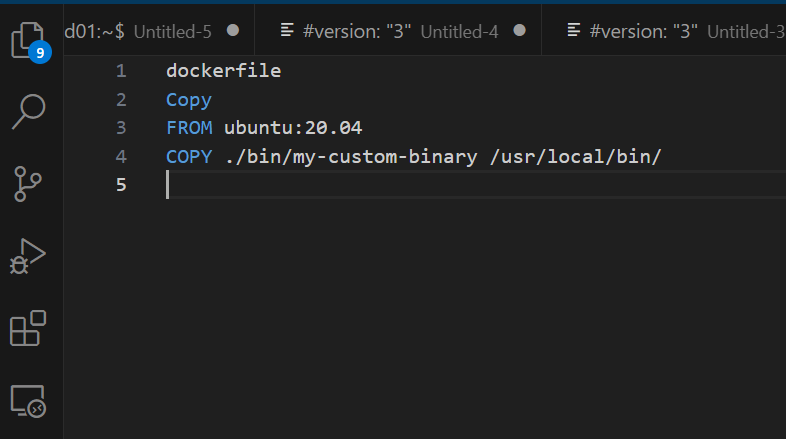
Once the external resources are identified and stored, they need to be integrated into the deployment process through the CI/CD pipeline (GitLab in this case). Here's how you would incorporate the handling of external resources in the pipeline:

1. **Fetching External Resources in CI/CD Pipeline**:
   * Modify the **GitLab CI/CD pipeline** (.gitlab-ci.yml) to include steps that fetch external resources from their respective locations (Persistent Volumes, ConfigMaps, Secrets, Artifact Repositories, or Cloud Storage).

Example (Fetching ConfigMap or Secret):



1. **Containerize with External Resources**:
   * Ensure that your Dockerfile or build configuration includes the logic to copy or download the required external files and binaries during the build phase.
   * Example Dockerfile for custom binaries:



1. **Pipeline Deployment Step**:
   * During the deploy step, ensure that the resources are properly mounted or injected into the application container. This can be done through Kubernetes volumes and volumeMounts for ConfigMaps, Secrets, or files from object storage.
2. **Version Control for External Files**:
   * Track external resources (such as configuration files) and their versions in GitLab (or a separate repository if needed). Ensure that the versions of the files correspond with specific versions of the application code.
   * Example: Use Git tags or branches to manage external resources versions.

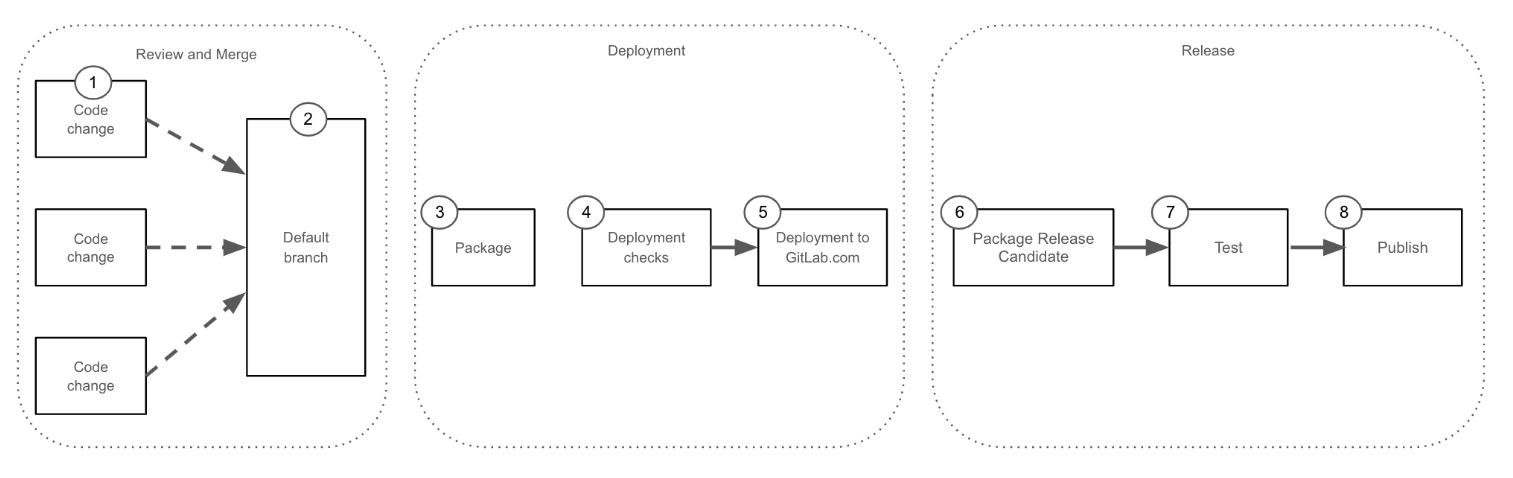
**Step 4: Security Considerations**

* **Secrets Management**:
  + Use **Kubernetes Secrets** or a tool like **HashiCorp Vault** for securely managing sensitive resources, such as database passwords, API keys, or access tokens.
  + Ensure that only authorized services/pods can access sensitive information using Kubernetes RBAC and Secret management tools.
* **Access Control**:
  + For external resources, implement **IAM roles** (in cloud platforms) or **RBAC policies** (in OpenShift/Kubernetes) to control access to persistent volumes, object storage, and other external resources.
* **Audit and Logging**:
  + Ensure that all access to external resources (e.g., object storage access or pulling from an artifact repository) is logged for auditing and debugging purposes.

**Step 5: Deployment to OpenShift**

* **Deployment Pipeline**:
  + Once the external resources are integrated into the pipeline and the necessary configurations are set, deploy the application using OpenShift’s oc apply, oc rollout, or Helm if you're using it for deployment management.
* **Monitor Resource Usage**:
  + After deployment, monitor the usage of external resources (e.g., storage utilization, file access, etc.) using OpenShift’s monitoring tools such as Prometheus and Grafana.

**This overview shows how the two processes are connected:**



**All Steps are explaining below:**

1. Engineer creates features or bug fixes. Changes reviewed by Maintainers
2. Validated changes merged into the default branch
3. A scheduled pipeline packages all new changes into an “auto-deploy package” for deployment to GitLab.com. Multiple packages are created each day at the listed times
4. If deployments are allowed the auto-deploy pipeline starts. Production Change Locks, unhealthy environments, or other ongoing deployments are examples of events that would prevent a deployment
5. The auto-deploy package is deployed to GitLab.com. For more details see the deployment process
6. Changes that have been successfully deployed to GitLab.com can be considered for packaged release for self-managed users. A new release candidate package is created for these changes
7. The release candidate is deployed to a test environment and automated QA tests execute
8. Release Candidate is officially tagged and published for release

**Design Principles**

**Development Guidelines**

* Playbooks, roles, and collections are developed with an idempotent end state in mind.
* Naming standards as well as code guidelines shall be followed as described under the Best Practices guide below.
* Projects should consist of their own structure and adhere to the following standard.
* A project folder should contain folder structure for the collections, roles, inventory, app\_configure.yml, and site yml
* Develop playbooks with major operations as tasks referenced using the site yml playbook.

1. Provision
2. Configure
3. Test
4. Remove

* Development complexity should be kept to a minimum. Adding additional modules adds complexity, which could potentially increase maintanence costs, and technical debt
* Default playbooks should consist of nothing more than executing a role's required tasks.
* Ansible roles need to be self contained and limit dependencies on other roles or external variables. See the "Roles" section below for more details.
* Variables shall follow a scheme of bracket notation instead of dot notation.
* Variable names shall be in snake\_case rather than camelCase.
* Ansible role logic should use assert with a fail or success message for conditionals
* Releases should target to deliver all portions of an Ansible collection concurrently
* Any deviations from the coding standard need to be discussed with the product owner

**Ansbile Inventory Guidelines**

* Ansible Inventories are used to define the footprint and machines for a deployment as well as collect required variables
* Galaxy inventories are structured to best simplify user customization
* The inventories exist as a single source of truth for desired configuration
* The inventories use the group\_vars folder to ensure groups and indvidual vars stay consistent despite changes to the target environment

**Best Practices Guide**

https://redhat-cop github.io/automation-good-practices/

**README Requirements**

* Every Galaxy code repositort will include a README with important context for utilizing the code
* The Galaxy teams plans to implement autogenerated READMES
  + READMES will include content for dependencies, versions, use-cases, and what is expected from the user for deployment of the collection or role

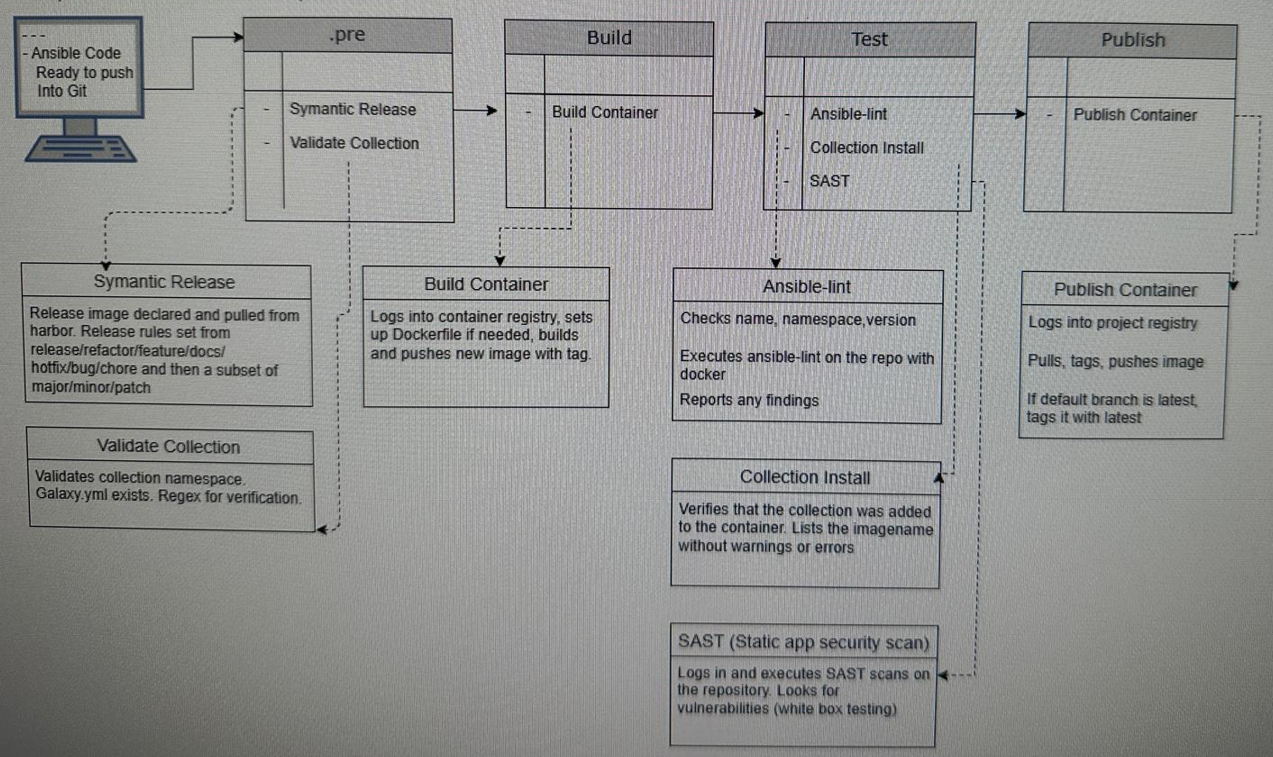
**Recommended Development Environment**

* Microsoft Visual Studio Code (VSCode)
* Red Hat Ansible Plugin https://marketplace visualstudio.com/items?itemName-redhat ansible
* Ansible-lint installed
* Configure the GitLab server to be able to push/pull content from VSCode

**Galaxy Code Development Pipeline**

**GitLab code ingestion pipeline**

When code is pushed into GitLab, the CI/CD pipeline is started and goes through the process of validations to determine pass/fail



**Common Codebase Repository Structure**

Git and GitLab will be used as the main source control for all automation code. Folder structure should be followed for documentation purposes, ease of use, and consistency.

Each folder from top level down will consist of a README file. The README file will be different, depending on the contents of the folder.

**1. Base Project collection level README**: (follow instructions at docs/developer/collections.md)

**• Description of the project/collection**

Author information

How to install the full collection

Roles listed inside the collection

Compatibility

License

**2. Images/Diagrams README**

List the images/diagrams that are in the current folder

**3 Roles README**

Name of role

Author information

Requirements

Dependencies

**Versions**

Example usage of the role

**License**

**4. File Docs README**

Description of the current folder

Files that exist in the folder

Ansible-lint default file should be present in folders that contain yaml Pass/fail for the lint check can be known before pipeline push.

**GitLab Standard CI/CD Variables**

The following CI/CD variables have been added to the Galaxy group and are available to all subgroups / projects.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | **Name** | **Purpose** | **Type** | | **Ansible-lint Configuration** | Ensure Ansible-lint default file is present in folders containing YAML files. Pass/Fail for the lint check can be known before pipeline push. | - | | **ASAP TOKEN** | Token used when executing the SAST (Static Application Security Testing) pipelines. | String | | **EFOSS TOKEN** | Token used in combination with EFOSS USER to get read access to Nexus repositories. | String | | **EFOSS USER** | User to use when pulling from Nexus repositories. | String | | **ENABLE\_SEMANTIC\_RELEASE** | When set to true, Semantic Release pipelines will execute. (true/false) | String | | **PROJECT\_PROTECTED\_BRANCH\_REGEX** | Regex used to setup a repository's protected branches. This can be customized on individual repositories to allow custom branch names. | String | | **PROJECT\_PROTECTED\_BRANCH\_USERS** | List of users allowed to push to protected branches (reference GitLab Protected branches API documentation for correct format). | String | | **REGISTRY** | Path to the Galaxy project in Harbor. | String | | **REGISTRY PASSWORD** | Password used with REGISTRY\_USER to push images to Harbor. | String | | **REGISTRY USER** | Username with access to push images into the Harbor container registry. | String | |  |  |  | |

**Ansible Playbook, Role, Collection, and Variable Guidelines**

This section describes how the LMCO Galaxy Project will design and develop customized Ansible automation code in order to make it understandable, reusable, and functional. Any co to be contributed to the LMCO Galaxy Collection is expected to meet the following standards and guidelines before being accepted.

The following structure definition and rules are meant to make it easy to group similiar capabilities into larger collections that can be packaged and made available to end-users. The structure should allow for growth and expansion of deployable capabilities with no, or minimal, rework. The namespacing structure should allow for a clear relationship to our git structur that matches the codebase This supports quick recognization of where a component resides for development and troubleshooting efficiency

**Ansible Facts**

Retreiving facts should be used sparingly, as there is a performance cost utilizing them.

As a developer, this should be declared in the README documentation for a role/collection either gather facts: true of as gather facts: false. If facts are required to use a role, then the fole should utilize the fall feature with a descriptive error if the necessary facts are not defined.

Facts blueprint: https://docs.us imco.com/display/ATB/Ansible+Facts+Blueprint

**Variables**

The LMCO Galaxy Project should use parameterized variables to customize deployments for end users.

Variable names shall follow standard Python naming conventions and use snake\_case\_naming for all YAML, Python files, script, variables, arguments, and repositories

Do not use special characters other than underscore

Use mnemonic and descriptive names for variables that would be considered human readable

Use standard patterns ex role/collection\_object\_<feature>\_<action>

Allow variables to be overridden

Set environment variables such as key-pairs to variables

Name all tasks, plays, and blocks to improve readability.

Use imperative names, such as "Ensure service is running", to show the action of the task while it's running

The master dictionary will be defined of different data structures that will be used as parameters. When passing information into roles/collections, use the dictionary parameters and create local variables to not change the global set, but also to limit the scope to the role. https://docs.us Imco.com/display/ATB/Parameters+Working+Session

**Playbooks**

The LMCO Galaxy Project will use playbooks as entry points to trigger developed Ansible Role content

The project will attempt to limit the size and count of individual playbooks by creating structured, reusable roles that are triggered by opinionated playbooks

The project will also use playbooks as part of the packaged collections to defined opinonated, full system builds

Playbooks should read like a workflow of tasks from a role for deploying a capability for each discrete installation method

Playbooks should use the include\_role task, rather than using the roles section. This can allow for the ability to add in control statements for more granular execution

**GitLab code merge approval**

As new code is pushed from a branch into main, it needs to go through an approval process to make sure it will merge and passes standards

