Reference Architectures 2017

Application CI/CD on OpenShift Container Platform with Jenkins

Last Updated: 2018-01-31

Reference Architectures 2017 Application CI/CD on OpenShift Container Platform with Jenkins

Aaron Weitekamp

Joseph Callen

Pep Mauri
refarch-feedback@redhat.com

Legal Notice

Abstract

The purpose of this document is to provide guidance using Jenkins with OpenShift Container Platform to continuously test and deploy applications.

Table of Contents

COMMENTS AND FEEDBACK **4**

CHAPTER 1. EXECUTIVE SUMMARY **5**
1.1. PROBLEM 5
1.2. LIMITATIONS 5
1.3. OPENSHIFT CONTAINER PLATFORM 3.7 5
1.4. LOCAL CLIENT TOOLS 5
1.5. OPENSHIFT PROJECTS 6

CHAPTER 2. COMPONENTS AND CONSIDERATIONS **7**
2.1. COMPONENTS 7
2.1.1. Software Version Details 7
2.1.2. OpenShift Clusters 7
2.1.3. Registry 7
2.1.4. Jenkins 8
2.1.5. Declarative Pipelines 8
2.1.6. Configuration 8
2.1.7. Developing Automation 9
2.2. GUIDING PRINCIPLES 11
2.2.1. Automate everything 12
2.2.2. Use appropriate tools 12
2.2.3. Parameterize Everything 12
2.2.4. Manage appropriate privilege 12

CHAPTER 3. ARCHITECTURAL OVERVIEW **14**
3.1. IMAGE PROMOTION PIPELINE WITH CENTRALIZED REGISTRY 14
3.2. USING IMAGE TAGS THROUGH THE SDLC 15
3.3. PIPELINES AND TRIGGERS 16
3.3.1. app-pipeline 16
3.3.2. release-pipeline 16
3.3.3. jenkins-lifecycle 17
3.3.4. app-base-image and jenkins-base-image 17
3.4. TOPOLOGY 17
3.5. CONFIGURATION VIA INVENTORY 18
3.5.1. Ansible Roles 18
3.5.1.1. auth 18
3.5.1.2. jenkins 18
3.5.1.3. puller 18
3.5.1.4. pusher 18
3.5.2. Playbook overview 18

CHAPTER 4. DEPLOYING THE AUTOMATION **20**
4.1. INITIAL CONFIGURATION 20
4.1.1. Variables 21
4.1.1.1. group_vars/all.yml 21
4.1.1.2. group_vars/[group].yml 22
4.1.1.3. host_vars/[environment]-1.yml 23
4.1.2. Example setups 23
4.1.2.1. TLS/SSL certificate validation 23
4.1.2.2. Single cluster / shared clusters 24
4.2. CUSTOMIZING THE AUTOMATION 25
4.2.1. Access to the projects 25

4.2.2. Application 25
4.2.3. Jenkins 25
4.3. EXECUTE AND MONITOR JENKINS APP-PIPELINE 26
4.4. EXECUTE AND MONITOR JENKINS RELEASE-PIPELINE 30

CHAPTER 5. CONCLUSION **34**

APPENDIX A. CONTRIBUTORS **35**

APPENDIX B. KNOWN ISSUES **36**

APPENDIX C. OPENSHIFT TEMPLATE MODIFICATIONS **37**

APPENDIX D. REVISION HISTORY **41**

COMMENTS AND FEEDBACK

In the spirit of open source, we invite anyone to provide feedback and comments on any reference architecture. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers. Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

CHAPTER 1. EXECUTIVE SUMMARY

Much has been written about continuous integration and delivery. It is impossible to deliver a “one size fits all” approach to such a complex topic. The scope of this paper is to provide guidance for mid-size teams using OpenShift to continuously test and deploy applications. Specifically, the focus is managing multi-cluster authentication and authorization, promoting applications between clusters and orchestrating the pipeline using an integrated Jenkins server.

For the purposes of this paper, assume a mid-size development team consisting of 12-50 people. Smaller teams may require less formalization than this paper outlines. Larger teams tend to have more centralized resources and more prescriptive processes that prevent adopting the recommendations this paper outlines. Limiting the scope of this paper highlights best how OpenShift addresses problems, including teams that are part of a much larger organization.

1.1. PROBLEM

Development teams want to move faster than traditional infrastructure, testing, and processes allow. OpenShift is designed to enable teams to adopt a culture which moves at a faster pace. When prioritizing automation in the development process, code is better and the team is more satisfied with the output. OpenShift is designed to be an integral part of automating the software development lifecycle (SDLC).

1.2. LIMITATIONS

This work is meant as guidance only. Red Hat will not provide support for the solution described in this document or source repository.

This Reference Implementation requires a few things to be set up properly, outlined in the following chapters.

1.3. OPENSHIFT CONTAINER PLATFORM 3.7

All commands, playbooks, and configurations require OpenShift Container Platform 3.7 and the Jenkins image included in that release.

There are multiple reference architectures available for OpenShift deployment on various cloud providers and on-premise infrastructure. Any of them may be utilized, including **`oc cluster up`** and the Red Hat CDK once updated with OpenShift 3.7 release.

1.4. LOCAL CLIENT TOOLS

In order to run Ansible playbooks, the OpenShift command line tools, git, and Ansible need to be available locally.

Table 1.1. Required local software

Software	RPM	Repository
git	git-1.8.3.1-12.el7_4.x86_64	rhel-7-server-rpms
ansible	ansible-2.4.1.0-1.el7.noarch	rhel-7-server-ose-3.7-rpms

Software	RPM	Repository
oc command	atomic-openshift-clients-3.7.9-1.git.0.7c71a2d.el7.x86_64	rhel-7-server-ose-3.7-rpms

1.5. OPENSHIFT PROJECTS

The lifecycle project on the dev cluster will run an instance of Jenkins. Each cluster and lifecycle project will run an instance of the application and database. Below are the PersistentVolumes required for each deployment type.

Table 1.2. Required PersistentVolumes

PersistentVolume	min.size	used for	used by
jenkins	2Gi	Jenkins Configs	Jenkins pod
mongodb	1Gi	Mongo Database	MongoDB pod

CHAPTER 2. COMPONENTS AND CONSIDERATIONS

This chapter describes the components required to setup and configure Application CI/CD on OpenShift 3.7. It also provides guidance to develop automation, based on experiences developing the examples in this Reference Implementation.

2.1. COMPONENTS

2.1.1. Software Version Details

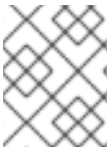
The following table provides installed software versions for the different instances comprising the reference implementation.

Table 2.1. Application CI/CD software versions

Software	Version
atomic-openshift{master,clients,node,sdn-ovs,utils}	3.7.9
Jenkins (provided)	2.73.3
ansible	2.4.1.0

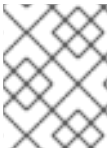
2.1.2. OpenShift Clusters

Multiple deployment clusters give teams flexibility to rapidly test application changes. In this reference implementation, two clusters and three OpenShift projects are used. The production cluster and project is separate from the non-production cluster. The non-production cluster is assigned to specific lifecycle projects: development and stage.



NOTE

The Ansible inventory, playbooks, and roles were written in a manner to support using any combination of clusters or projects.



NOTE

Both the Jenkins OpenShift client and Ansible use the **oc** client. If network connectivity is available between clusters location should not matter.

2.1.3. Registry

OpenShift deploys an integrated registry available to each cluster. In this reference implementation, an OpenShift-based external registry to the clusters is used. This enables a more flexible application promotion and cluster upgrade process. To promote an application, the target cluster pulls the application image from the external registry, simplifying multi-cluster coordination. When a cluster must be upgraded, the critical application image remains on an independent system.

NOTE

Additional external registries may also be utilized including Sonatype Nexus, JFrog Artifactory, or an additional project within an existing OpenShift cluster.

2.1.4. Jenkins

Development teams need a service to drive automation but want to minimize the effort required to configure and maintain an internal-facing service. The integrated Jenkins service addresses these concerns in several ways:

- Authentication
- Authorization
- Deployment configuration
- Pipeline integration
- OpenShift plugin

These integrations simplify Jenkins server operation so a team can focus on software development. By integrating **authentication**, teams are much less likely to use shared server credentials, a weak backdoor administrative password, or other insecure authentication practices. By integrating **authorization**, team members inherit the same OpenShift project privilege in the Jenkins environment, allowing them to obtain necessary information while protecting the automation from unauthorized access. By stabilizing **deployment configuration**, teams have a straightforward way to store Jenkins configurations in source control. For example, instead of adding plugins through the web user interface, a Jenkins administrator may make a change to a simple text file managed by source control. This source control change can trigger a redeployment of the Jenkins server. With this model, the deployment is now reproducible and plugin failures may be quickly rolled back. Jenkins **Pipeline integration** allows users to view pipeline status directly from the OpenShift web UI for an integrated view into the deployment lifecycle. The Jenkins **OpenShift Client plugin** allows team members to more easily automate calls to OpenShift, simplifying pipeline code.

See Jenkins image documentation for reference.

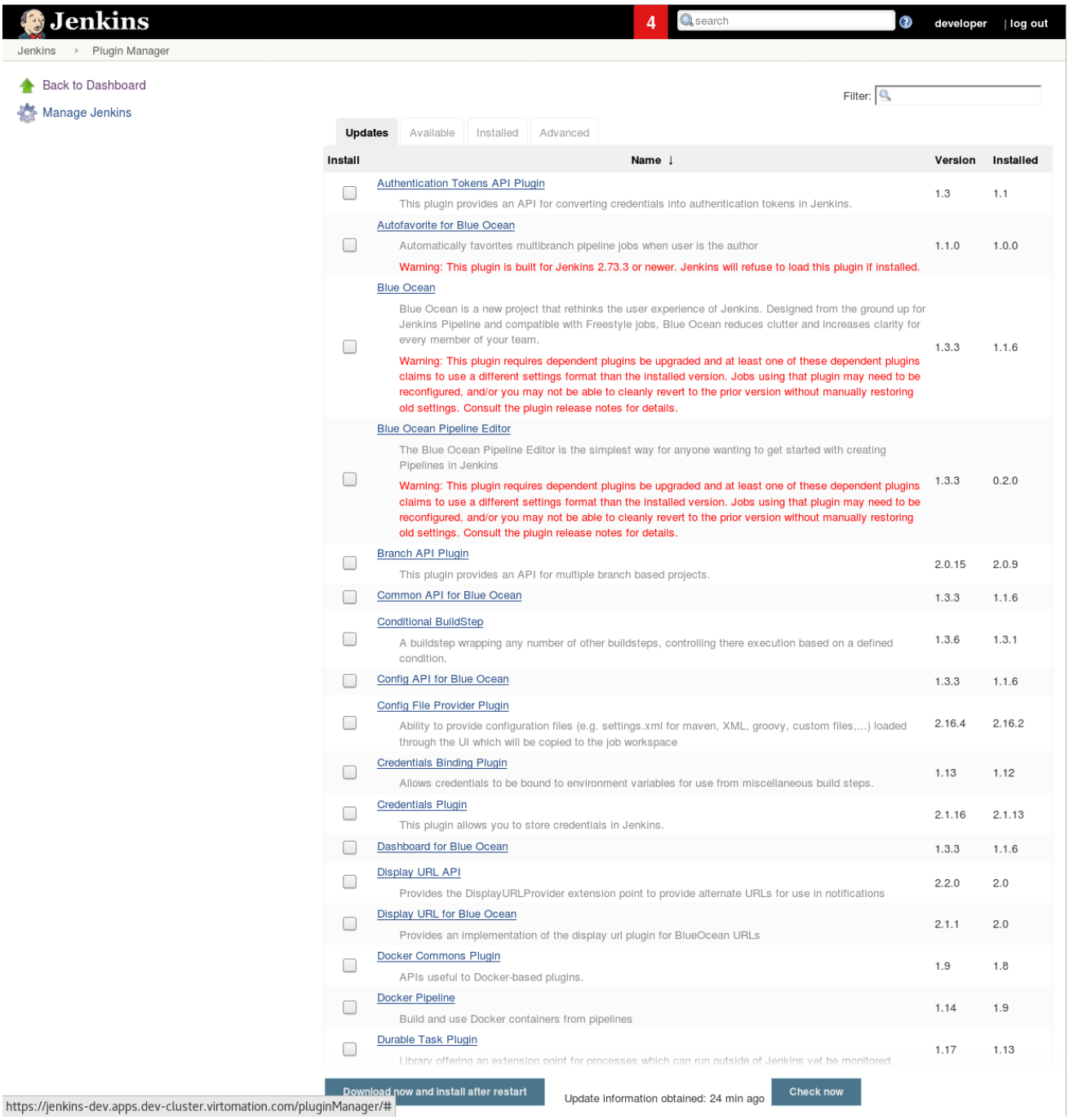
2.1.5. Declarative Pipelines

The Jenkins declarative pipeline syntax is a relatively recent approach to defining workflows in Jenkins. It uses a simpler syntax than the scripted pipeline. Using this syntax allows integration with OpenShift. In this project, the declarative pipeline syntax is used exclusively.

2.1.6. Configuration

Maintaining Jenkins configuration in source control has several benefits but exposes some special cases as well. A good practice is pinning plugin versions using a plugins.txt file (see the section about Jenkins customization for an example) so plugins are not arbitrarily updated whenever a new Jenkins configuration is deployed. In this case, however, more diligence must be exercised to keep plugins updated. Refer to the Jenkin master plugin management page (**[Manage Jenkins]** → **[Manage plugins]**) to understand which plugins must be updated.

Figure 2.1. Jenkins Update Center



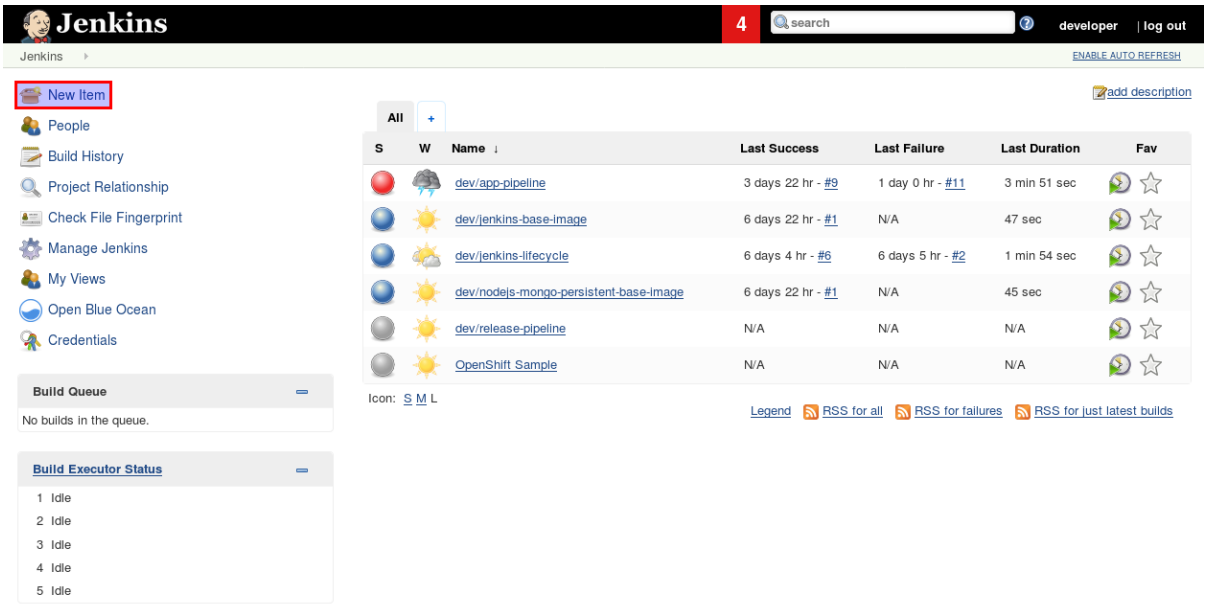
2.1.7. Developing Automation

Developing Jenkins automation can be a frustrating experience. In many cases, the pipeline files are downloaded from source control each time the pipeline runs. This leads to development cycles where a source control commit is made to test out a change. Also, the code is not easily run on a local workstation.

One development environment option is to use the pipeline sandbox to create pipeline code directly in the Jenkins web interface. Development cycle time is greatly reduced and an interactive step generator is available to assist with the specific plugin syntax. To use this environment:

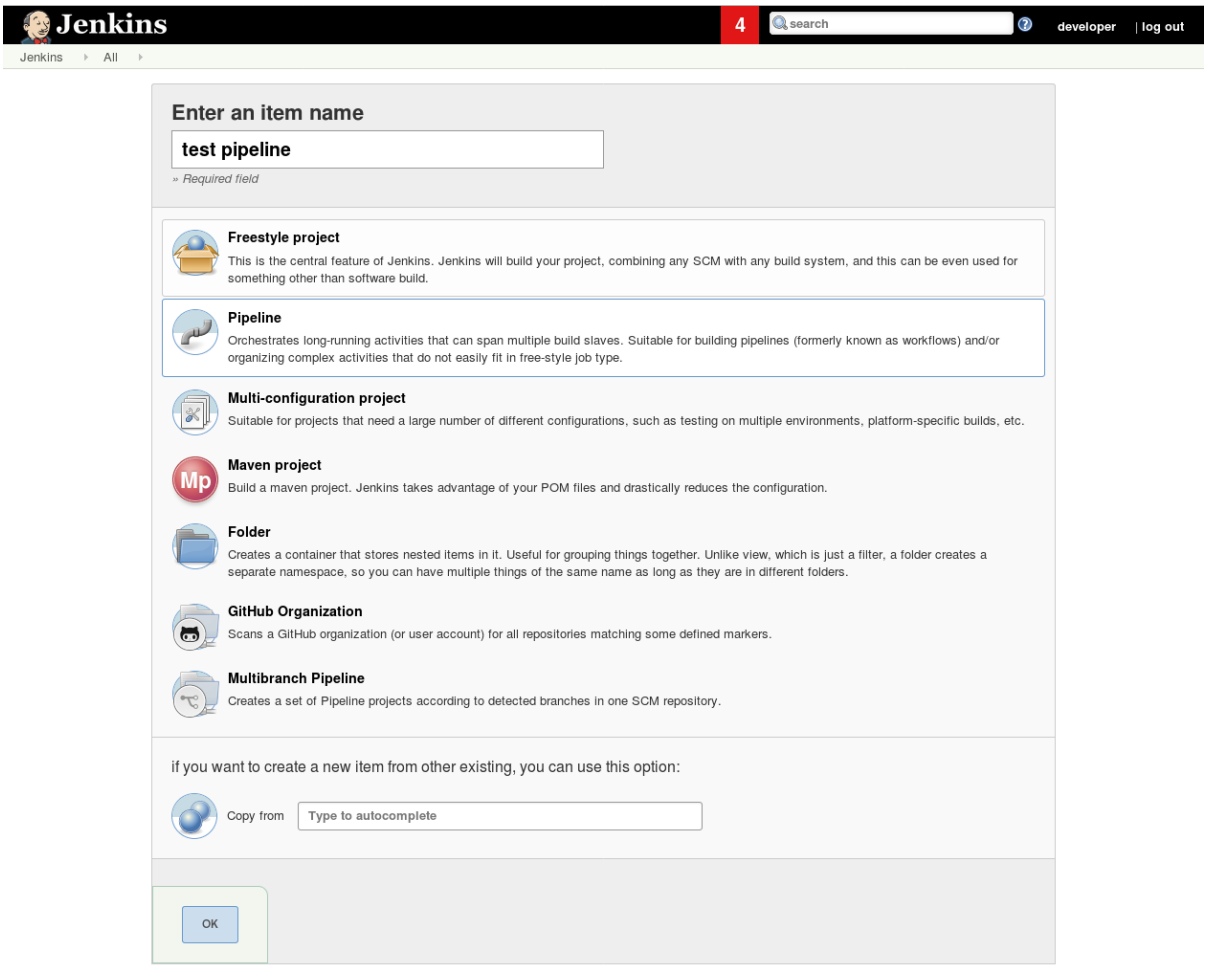
1. From the Jenkins home dashboard, select **[New item]**.

Figure 2.2. Jenkins Dashboard



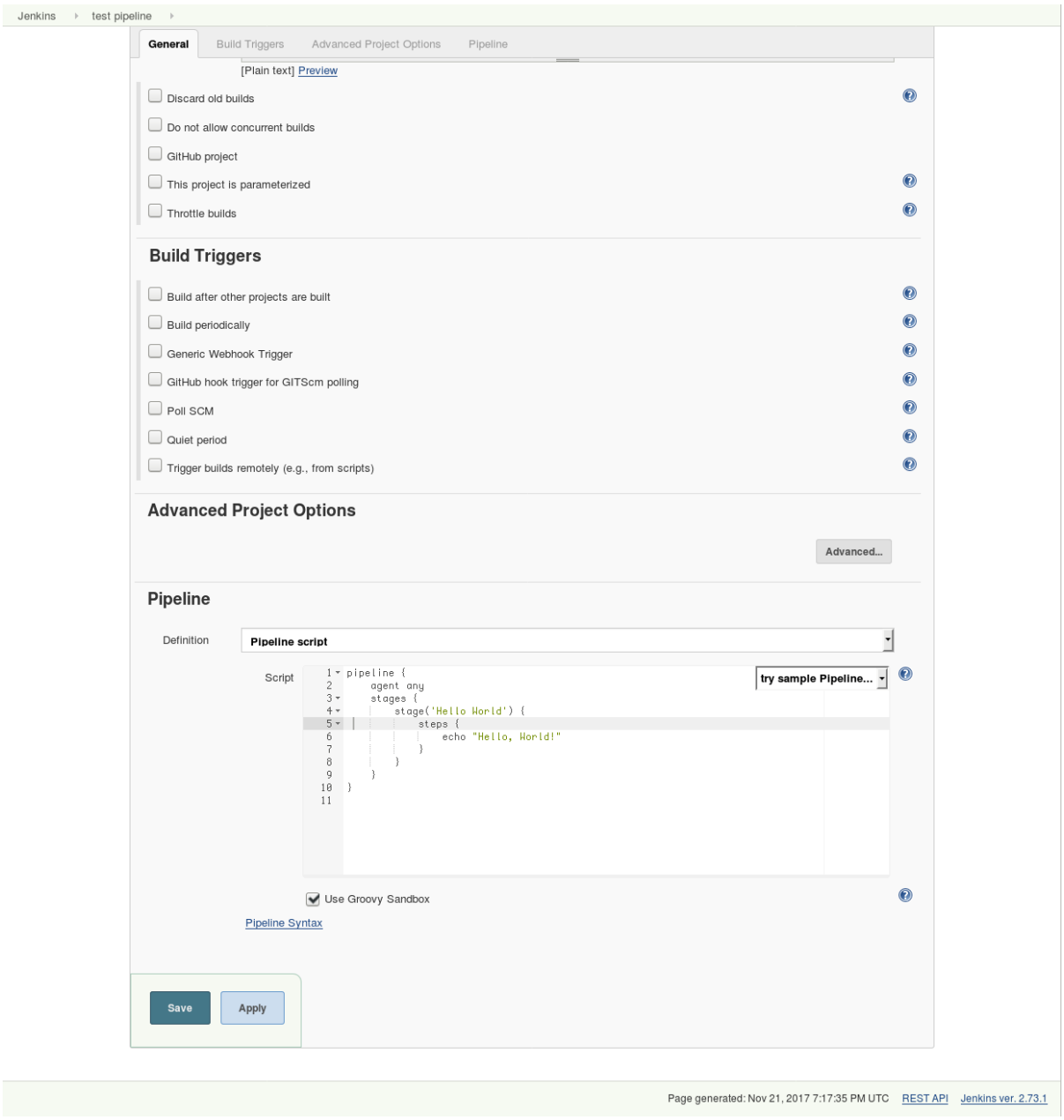
2. Name the test pipeline, select **[Pipeline]** project type and click **[OK]**.

Figure 2.3. Jenkins New Item



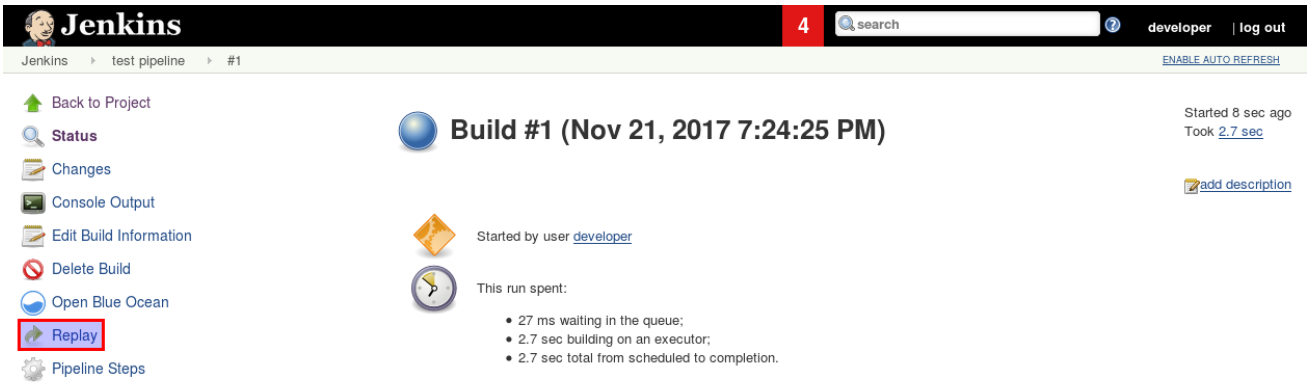
3. Edit the pipeline directly in the **[Pipeline]** text area. For snippet generation and inline plugin documentation, click **[Pipeline Syntax]**.

Figure 2.4. Jenkins New Pipeline



Once a pipeline has been committed to source control, Jenkins allows a user to edit a previously run pipeline and **[Replay]** the task.

Figure 2.5. Jenkins Pipeline Replay



2.2. GUIDING PRINCIPLES

2.2.1. Automate everything

The OpenShift web interface and CLI are useful tools for **developing**, **debugging** and **monitoring** applications. However, once the application is running and basic project configuration is known, these artifacts should be committed to source control with corresponding automation, such as Ansible. This ensures an application is reproducible across clusters and project configuration changes can be managed, rolled back, and audited.

The important discipline is to enable automation to be run at any time. In this project, the ansible playbook may be run arbitrarily to keep the system in a known state. This ensures the environments and source-controlled configuration do not drift. Running the playbook on a regular basis provides the team a limited set of changes between runs, which makes debugging much simpler. If drift is suspected, the automation becomes effectively useless.

2.2.2. Use appropriate tools

Many automation tools are able to solve similar problems. Each tool has a unique way of solving a particular stage of automation. The following questions can help make tooling choices:

- What stage of automation is involved?
- Who will be maintaining the automation?
- What user or process will be running the automation?
- How will the automation be triggered?

For this paper the following choices have been made:

- Ansible configures the OpenShift projects and authorization
- Jenkins provides centralized orchestration of the application lifecycle, including builds, tests, and deployments

2.2.3. Parameterize Everything

Parameterization enables a team to centralize the configuration of a project so changes involve less risk. Project values may need to be altered for testing, migrating to a new cluster, or due to a changing application requirement. Since Ansible provides the baseline configuration for the cluster, including Jenkins, playbook parameterization is a good place to define parameters shared across clusters.

Using this principle, most parameter values for this paper flow through the system in this way:

Ansible → OpenShift → Jenkins

This requires parameters to be mapped in several places. For example, if parameters are defined using Ansible group and host variable files, the OpenShift objects created must be templated to replace these values. Regardless, the benefits are considered worthwhile.

For this project, we have chosen to prefer parameterized OpenShift templates over Jinja2-style Ansible templates. This allows the OpenShift templates to be used without Ansible.

Keeping a consistent parameterization approach simplifies maintenance and reduces confusion.

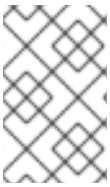
2.2.4. Manage appropriate privilege

Team members should have unrestricted access to necessary debugging information required to perform job functions effectively. Source control should also be relied upon to trigger state change as much as possible. With this model, the team will be encouraged to automate everything while maintaining access to information needed to resolve problems. The following table provides privilege suggestions for a team:

Table 2.2. Suggested Project Privileges

	Registry	Dev cluster	Stage cluster	Prod cluster
Dev leads	pull only	project administrator	project editor	project viewer, run debug pods
Developers	pull only	project editor	project editor	project viewer
Operators	pull only	project viewer	project editor, varied cluster priv	project editor, varied cluster priv

These are suggestions to stimulate conversation. Privilege granularity may vary widely depending on the size and culture of the team. Appropriate cluster privilege should be assigned to the operations team. What is important is to develop a privilege plan as a DevOps team.



NOTE

Since team membership and roles change over time, it is important to enable unassigning privilege. This reference implementation includes a way to manage lists of deprecated users. See the group variables for an example.

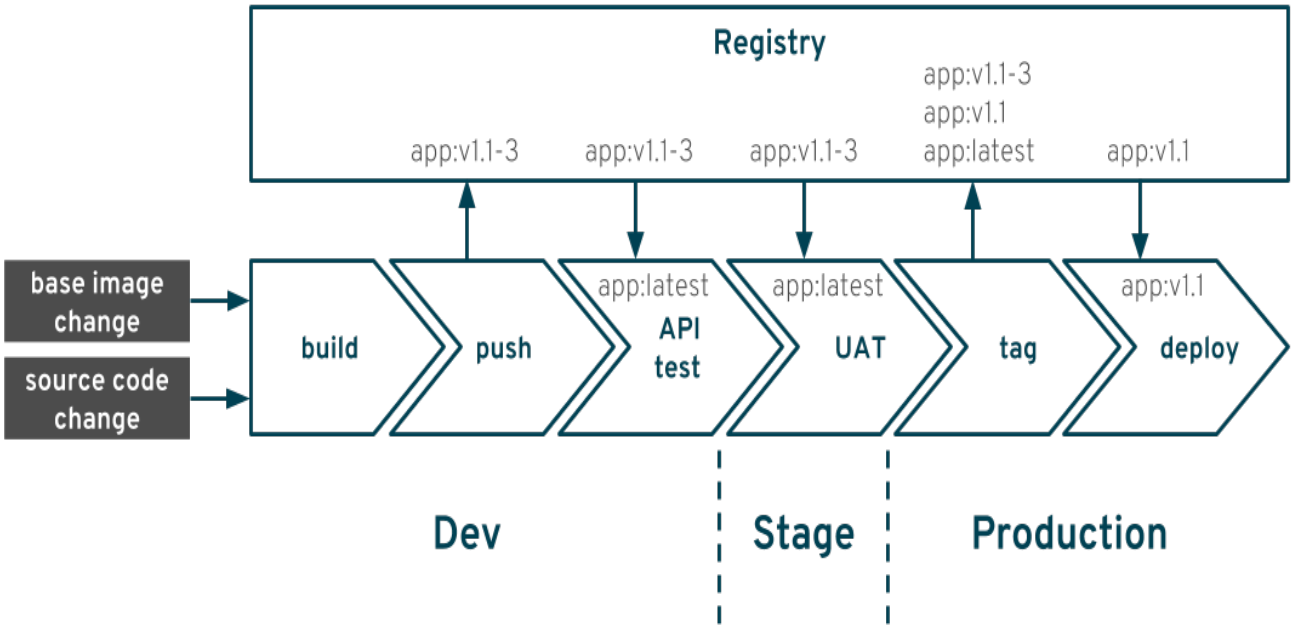
An authentication token is used in this reference implementation to allow deployment orchestration in each cluster from the Jenkins master. The **jenkins** token is mounted as a secret in the Jenkins pod. Jenkins uses the credential plugin to securely manage cluster access and protect secrets from leaking into log files.

CHAPTER 3. ARCHITECTURAL OVERVIEW

3.1. IMAGE PROMOTION PIPELINE WITH CENTRALIZED REGISTRY

Image tagging and imagestreams are used to track build deployments across the lifecycle environment. The following high-level diagram follows a single image through the application lifecycle.

Figure 3.1. Pipeline Promotion



1. The pipeline may be triggered by either:
   a. A base image change, e.g. registry.access.redhat.com/openshiftv3/nodejs-mongo-persistent.
   b. A source code commit.
2. An OpenShift build is initiated.
3. The resulting container image is pushed to the registry, tagged with the current version number plus a build increment, e.g. **`v1.1-3`**. The dev environment moves the **`app:latest`** imagestream tag to point to the new image in the registry **`app:v1.1-3`**.
4. A test deployment is run to test the functioning API, pulling the built image using the latest tag.
5. If passed, the stage environment moves the **`app:latest`** imagestream tag to point to the registry tag **`app:v1.1-3`**. Since the 'latest' imagestream tag has changed a new deployment is initiated to run the user acceptance test (UAT) phase.

The dev/stage pipeline ends at this phase. Pass/Fail may be posted to internal systems. Many builds may run through to this stage until a release candidate is identified. When a specific build is planned for release, an authorized person initiates the release pipeline.

1. An authorized user logs into OpenShift to initiate the release by providing the specific build number, e.g. **`v1.1-3`**.
2. The release candidate image is tagged in the registry as **`v1.1`** and **`latest`**. This convention ensures the registry is always serving the latest release with tag 'latest' and the released version

can also be pulled using the <majorVersion.minorVersion> format. The previously released version remains available as ‘v1.0’.

3. The production environment tags an imagestream to point to tag v1.1 in the registry, updates the deployment config and rolls out the application.

3.2. USING IMAGE TAGS THROUGH THE SDLC

Tags are the primary method for identifying images. They are mutable objects so they must be handled carefully. An OpenShift-based registry allows automation to remotely tag images without re-pushing.

There are several conventions used to help users and automation identify the correct image.

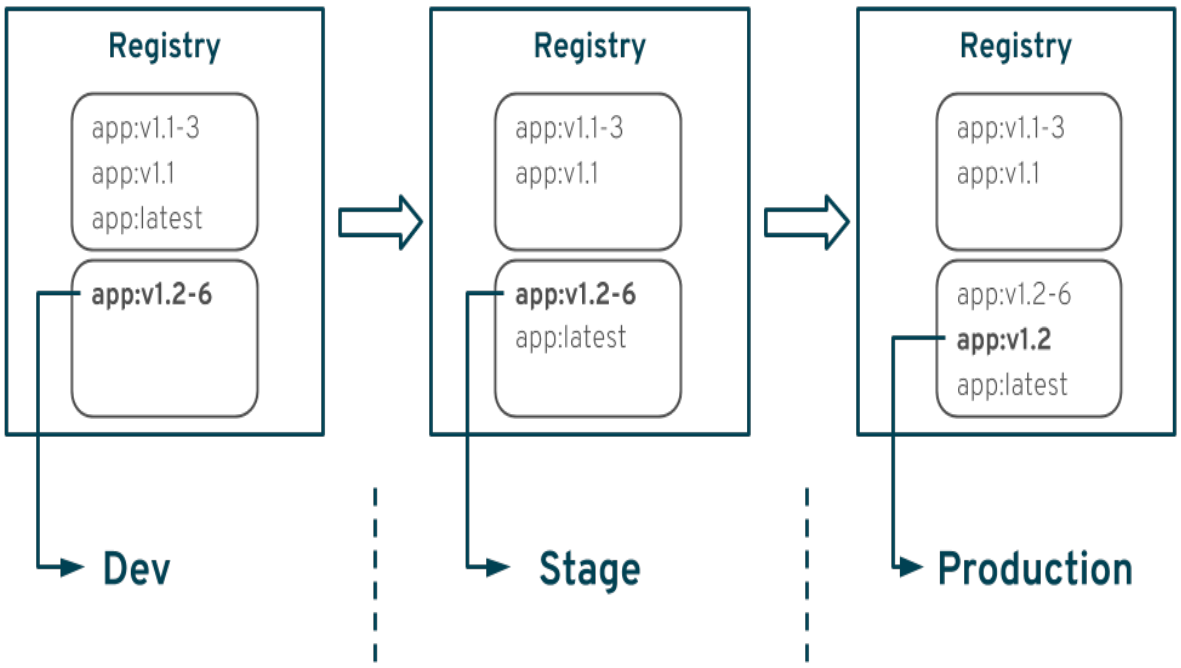
1. The ‘latest’ tag is a convenience to reference the latest image that has passed dev environment testing.

2. The application version, <majorVersion.minorVersion> references the latest *release* of that particular version.

3. A build number is appended to identify a specific image under test. These images take the form of <majorVersion.minorVersion-buildNumber>.

The following graphic depicts an example of how two images are tagged in a single registry throughout the SDLC. The **boldface** image tags identify which tag is referenced in the deployment.

Figure 3.2. Registry tagging



1. Test image tag ‘v1.2-6’ is pushed to the registry and deployed to the development environment. The ‘6’ is the Jenkins build number.

2. If tests pass in the development environment the image tag ‘v1.2-6’ is deployed to the stage environment.

3. If tag ‘v1.2-6’ is chosen for release, the image is tagged ‘v1.2’, which identifies this is the released version. The image is also tagged ‘latest’. Tag ‘v1.1’ is still available but it is no longer ‘latest’.

3.3. PIPELINES AND TRIGGERS

Both OpenShift and Jenkins provide methods to trigger builds and deployments. Centralizing most of the workflow triggers through Jenkins reduces the complexity of understanding deployments and why they have occurred.

The pipelines buildconfigs are created in OpenShift. The OpenShift sync plugin ensures Jenkins has the same pipelines defined. This simplifies Jenkins pipeline bootstrapping. Pipeline builds may be initiated from either OpenShift or Jenkins.

The following table describes the pipelines in this project.

Table 3.1. Jenkins Pipelines

Pipeline	Purpose	Trigger
app-pipeline	Manage app deployment across dev and stage clusters	Poll SCM
release-pipeline	Tag image for release and rollout to production environment	Manual
jenkins-lifecycle	Manage Jenkins Master	Poll SCM
app-base-image jenkins-base-image	Notify the application base image has changed	Base imagestream:latest change

3.3.1. app-pipeline

The app-pipeline is an example that manages the nodejs-ex application deployment across the projects or clusters. There are a number of stages to complete the promotion process described in Section 3.1, “Image Promotion Pipeline with Centralized Registry”, and the *app-pipeline* handles the Dev and Stage part. Let’s see an overview of the steps the example *app-pipeline* performs.

First, the stage cluster authentication is synchronized into Jenkins which will be used in a later stage. As previously mentioned, **`nodejs-ex`** is being used as the example application. Fortunately, it contains testing that can be reused, which is executed next. If testing passes then the application’s OpenShift template is processed and applied. The next stage is to build the image. Once complete the tag is incremented as described in Section 3.2, “Using Image Tags Through the SDLC”. After the image becomes available it can be rolled out. If that completes successfully, the process is duplicated for the stage cluster.

3.3.2. release-pipeline

The release-pipeline is very similar to the Section 3.3.1, “app-pipeline” and addresses the remaining steps in Section 3.1, “Image Promotion Pipeline with Centralized Registry”. Credentials for the production and registry clusters must be synchronized and the OpenShift template must also be applied.

The only major difference is prompting for the specific image tag to be promoted to production. This stage also performs the necessary tagging described in Section 3.2, “Using Image Tags Through the SDLC”. After the pipeline is complete, an email will be sent to the configured recipients.

3.3.3. jenkins-lifecycle

If there are changes within the source repository’s jenkins directory Jenkins will initiate a rebuild of the image with the appropriate modifications.

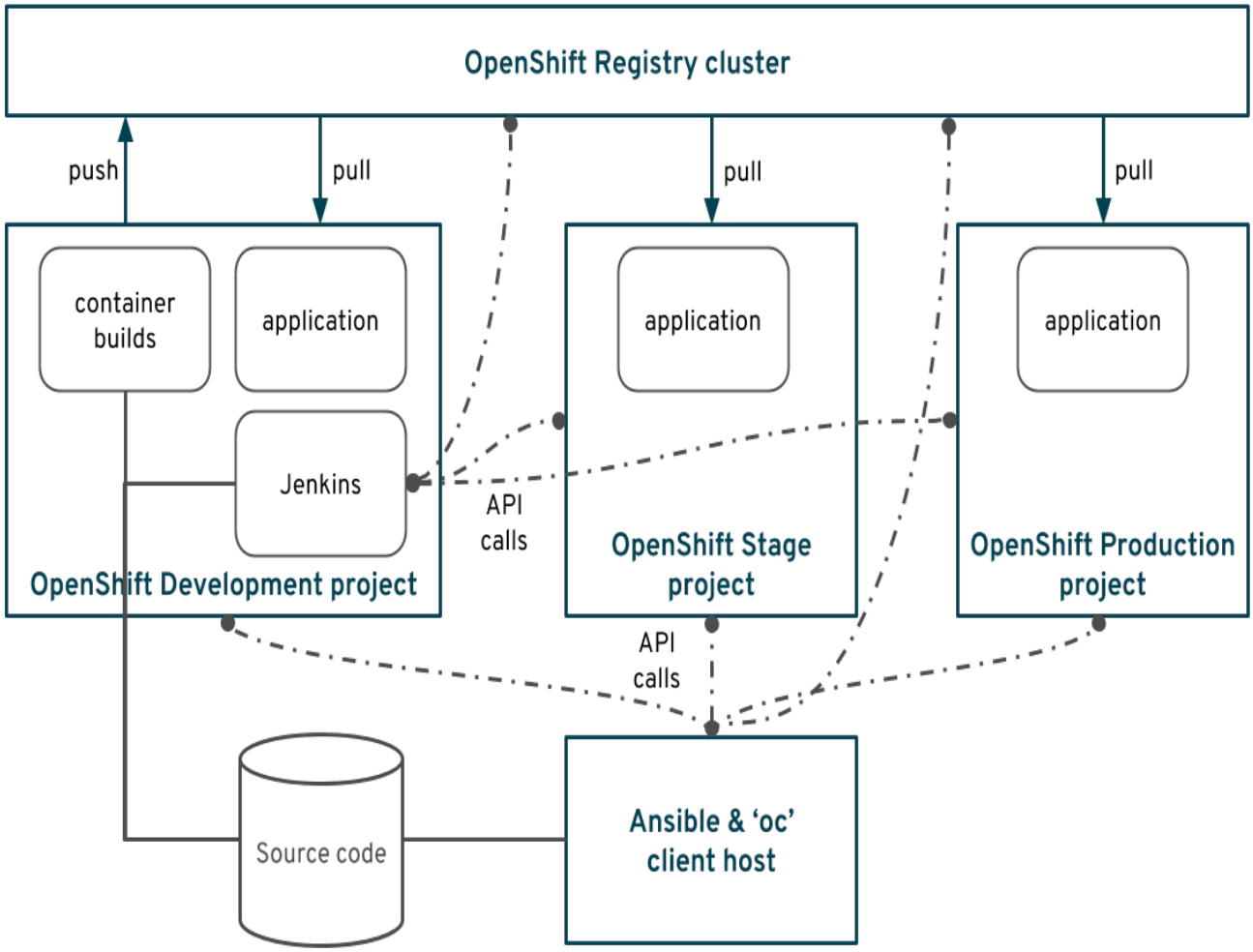
3.3.4. app-base-image and jenkins-base-image

The base-image-pipeline is embedded in an OpenShift BuildConfig and Template object. Using an OpenShift Template simplifies the deployment and configuration of multiple pipelines that will monitor base image changes. In the example configuration, the two pipelines created will monitor for updates to the Jenkins or nodejs source-to-image (S2I) images. When detected, the pipeline will prompt to upgrade which will trigger a Jenkins job to rebuild the image.

3.4. TOPOLOGY

This graphic represents the major components of the system.

Figure 3.3. Topology



All artifacts are in source control. These artifacts are used by **`ansible-playbook`** and **`oc`** remote clients. The clients deploy these artifacts to the development cluster via API. Jenkins deploys to the other clusters via API.

The dev cluster performs S2I container builds, pushing them to the registry. All environments pull these images to the clusters.

3.5. CONFIGURATION VIA INVENTORY

To reference multiple clusters (or projects) a specific Ansible inventory pattern may be used. Since OpenShift projects may be configured remotely, no SSH connection is required. In Ansible this is a “local” connection. An inventory file may be used that references arbitrary hosts all using the “local” connection. This means all commands will be executed on the local machine but the hostvars will cause the API calls to be made against the target cluster. In this example inventory file, four groups and hosts are defined: dev/dev-1, stage/stage-1, prod/prod-1 and registry/registry-1. Each group and host corresponds to an OpenShift cluster or project. The playbook can be executed against each individually using a specific group (e.g. "stage") or hostname "all" to apply a playbook against all clusters. The example group and host variable files provide group and host-specific connection details.

3.5.1. Ansible Roles

Ansible is designed to be self-documenting. However, additional explanation may be beneficial.

3.5.1.1. auth

- Configure service accounts.
- Bind and unbind roles for listed users and service accounts.
- Retrieve registry token for push and pull.

3.5.1.2. jenkins

- Create secrets on Jenkins host with tokens from other clusters.
- Start custom Jenkins build.
- Create Jenkins service.
- Create pipeline buildconfigs.

3.5.1.3. puller

- Create dockercfg secret so appropriate clusters may pull from a central registry.
- Link dockercfg secret with default service account.

3.5.1.4. pusher

- Create dockercfg secret so appropriate clusters may push builds to the central registry.
- Link dockercfg secret with builder service account.

3.5.2. Playbook overview

The reference Ansible playbook performs the following tasks:

1. Bootstrap the projects involved by creating the projects, if needed, and obtaining an authentication token for the admin users to each project to be used by subsequent **oc** client operations.

2. Configure authorization in all projects by invoking the Section 3.5.1.1, “auth” role.

3. Prepare the *dev* environment by setting up Jenkins through the Section 3.5.1.2, “jenkins” role and preparing registry push credentials through the Section 3.5.1.4, “pusher” role.

4. Prepare the *stage* and *production* environments by setting up image pull permissions using the Section 3.5.1.3, “puller” role.

CHAPTER 4. DEPLOYING THE AUTOMATION

The automation used in this reference implementation can be found here. The code should be checked out to the local workstation:

```
$ git clone https://github.com/RHsyseng/jenkins-on-openshift.git
$ cd jenkins-on-openshift/ansible
```

The repository contains a reference Ansible playbook in **`main.yml`** to configure a set of application project namespaces on a set of OpenShift clusters, covering an application's life-cycle through separate projects for development, staging, and production, using a shared container image registry.

TIP

When using the **oc** client to work against multiple clusters it is important to learn how to switch between contexts. See the command line documentation for information on cluster context.

The reference playbook deploys a Jenkins instance in the development cluster to drive the life-cycle of the application through the environments.

4.1. INITIAL CONFIGURATION

1. Configure the environments: using the *group_vars/*.yml.example* files, rename each file to remove '.example', e.g.

   ```
   for i in group_vars/*.example; do mv "${i}" "${i/.example}"; done
   ```

   The directory should look like this:

   ```
   .
   └── group_vars
       ├── all.yml
       ├── dev.yml
       ├── prod.yml
       ├── registry.yml
       └── stage.yml
   ```

   Edit these files and adjust the variables accordingly. At the very least, these settings must be customized (see the variables section below for more details):

   - **all.yml**: **`central_registry_hostname`**
   - **[group].yml**: **`clusterhost`**

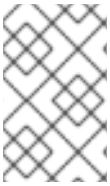
2. Configure authentication to each environment using the *host_vars/*-1.yml.example* files as a guide. Rename each file to remove '.example'. The directory should look like this:

   ```
   .
   └── host_vars
       ├── dev-1.yml
       ├── prod-1.yml
       ├── registry-1.yml
       └── stage-1.yml
   ```

Then edit each of the files and set the respective authentication information. See the host variables section for more details.

3. Run the playbook:

```
ansible-playbook -i inventory.yml main.yml
```



NOTE

A number of Ansible actions may appear as failed while executing the playbook. The playbook is operating normally if the end result has no failed hosts. See ignoring failed commands for additional information.

4.1.1. Variables

Below is a description of the variables used by the playbooks. Adjust the values of these variables to suit the given environment, clusters, and application.

The various variables are stored in different files depending on the scope they have, and therefore are meant to be configured through the group or host variable files in **`{group,host}_vars/*.yml`**.

However, Ansible's variable precedence rules apply here, so it is possible to set or override some variable values in different places.

For example, to disable TLS certificate validation for the staging environment/cluster only, **`validate_certs: false`** may be set in **`group_vars/stage.yml`** while also keeping **`validate_certs: true`** in the **`group_vars/all.yml`** file.

It is also possible to override values in the inventory file and via the **`--extra-vars`** option of the **`ansible-playbook`** command. For example, in a single cluster environment it may be better to set the value of **`clusterhost`** just once as an inventory variable (i.e. in the **`vars`** section of **`inventory.yml`**) instead of using a group variable. Moreover, if the same user is the administrator of all the projects, instead of configuring authentication details in each environment's file inside **`host_vars`** a single token can be passed directly to the playbook with:

```
ansible-playbook --extra-vars token=$(oc whoami -t) ...
```

See the Ansible documentation for more details.

4.1.1.1. `group_vars/all.yml`

This file specifies variables that are common through all the environments:

Table 4.1. Variables common through all environments

Variable Name	Required Review	Description
central_registry_hostname	Yes	The hostname[:port] of the central registry where all images will be stored.
source_repo_url	No	git repository URL of the pipelines to deploy

Variable Name	Required Review	Description
source_repo_branch	No	git branch to use for pipeline deployment
app_template_path	No	Relative path within the git repo where theapplication template is stored
app_name	No	Name of the application
app_base_tag	No	Base ImageStreamTag that the application will use
validate_certs	Yes	Whether to validate the TLS certificates during cluster/registry communications
notify_email_list	Yes	Email notifications from pipelines: destination
notify_email_from	Yes	Email notifications from pipelines: from
notify_email_replyto	Yes	Email notifications from pipelines: reply-to
oc_url	No	URL location of OpenShift Origin client
oc_extract_dest	Yes	Disk location that the client will be downloaded and extracted
oc_path	Yes	Path to OpenShift client (used if workaround is False)
oc_no_log	No	Disables logging of oc commands to hide OpenShift token.
enable_dockercfg_workaround	Yes	Implements workaround described in the Appendix Known issues

4.1.1.2. `group_vars/[group].yml`

There is a **`.yml`** file for each of the environments: development (**`dev`**), staging (**`stage`**), production (**`prod`**), and shared **`registry`**. Each of these files contains variables describing their respective cluster and project details:

Table 4.2. Variables common through all environments

Variable Name	Required Review	Description
clusterhost	Yes	Specifies the hostname[:port] to contact the OpenShift cluster where the environment is hosted. Do not include the protocol (**`http[s]://`**).

Variable Name	Required Review	Description
project_name	No	Describe the project where the respective environment is hosted.
project_display_name	No	Describe the project where the respective environment is hosted.
project_description	No	Describe the project where the respective environment is hosted.
{admin,editor,viewer}_{users,groups}	No	A set of lists of users/groups that need permissions on the project. Users listed in the **`users`** role get **`[role]`** permissions *granted*.
deprecated_{admin,editor,viewer}_{users,groups}	No	Add users/groups that must have their permissions *revoked*.

4.1.1.3. `host_vars/[environment]-1.yml`

These files contain authentication credentials for each of the environments.

Depending on the authentication method of the OpenShift cluster, authentication credentials can be provided either as **`openshift_username`** and **`openshift_password`** ***or*** as an authentication token.

A token (for example obtained from **`oc whoami -t`**) always works. Moreover, it is the only option if the available authentication method requires an external login (for example GitHub), where a username/password combination can not be used from the command line.

TIP

Since we are dealing with authentication information, ansible-vault can help protect the information in these files through encryption.

4.1.2. Example setups

Here are some sample values for the configuration variables to address specific needs.

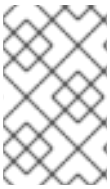
4.1.2.1. TLS/SSL certificate validation

The **`validate_certs`** variable is a Boolean that enables or disables TLS certificate validation for the clusters and the registry.

It is important to keep in mind the playbooks provided here only interact with the configured OpenShift clusters through an API, and do not interfere with the cluster's own configuration.

Therefore, if for any reason TLS certificate validation is disabled for a cluster, the cluster administrator must also take measures to ensure the cluster operates accordingly.

In particular, image push/pull is performed by the container runtime in each of the nodes in the cluster. If **`validate_certs`** is disabled for the registry being used (**`central_registry_hostname`**), the nodes also require the registry to be configured as an insecure registry.



NOTE

To configure a node to use an insecure registry edit either **`/etc/containers/registries.conf`** or **`/etc/sysconfig/docker`** and restart docker.

NOTE

Disabling certificate validation is not ideal; properly managed TLS certificates are preferable. OpenShift documentation has sections on Securing the Container Platform itself as well as Securing the Registry.

4.1.2.2. Single cluster / shared clusters

The playbook is designed to operate on four separate OpenShift clusters (one project on each), each hosting one of the environments: development, staging, production, registry.

It is possible to share the same cluster among various environments (potentially all 4 running on the same cluster, on separate projects) by pointing them to the same **`clusterhost`**. This is particularly useful during local testing, where the whole stack can run on a single all-in-one cluster powered by minishift or **`oc cluster up`**.

However, if the *registry* project shares a cluster with other projects extra care must be taken to ensure the images in the registry's namespace are accessible from the other projects sharing the cluster.

With regard to the ability of the other projects to use images belonging to the registry's namespace, relevant project's service accounts must be granted *view* permissions to the registry's project. This is achieved by adding their service accounts' groups to the **`viewer_groups`** in **`group_vars/registry.yml`** (see environment group_vars):

```
viewer_groups:
  - system:serviceaccounts:dev
  - system:serviceaccounts:stage
  - system:serviceaccounts:prod
```

NOTE

These are groups of service accounts, so it is **`system:serviceaccounts`** (with an *s*). Adjust the names according to the **`project_name`** of the respective environment.

If the **`dev`** project shares the same cluster with the **`registry`** project, one additional requirement exists. Images are built on this project, so its *builder* service account needs privileges to *push* to the registry's namespace. One way to achieve this is by adding that service account to the list of users with an *editor* role in **`registry.yml`**:

```
editor_users:
  - system:serviceaccount:dev:builder
```

4.2. CUSTOMIZING THE AUTOMATION

4.2.1. Access to the projects

As discussed in Section 2.2.4, “Manage appropriate privilege”, it is important to provide appropriate privilege to the team to ensure the deployment remains secure while enabling the team to perform their job effectively. In this reference implementation, the Ansible auth role provides tasks for managing authorization. Using host_vars and group_vars files we are able to manage project access across the clusters. The auth role may be simplistic but is easily extended using the provided pattern.

4.2.2. Application

The application managed by this CI/CD workflow is defined by two main configuration items:

- A *template* for the application’s build and deployment configurations
- A set of *parameters* to control the template’s instantiation process

These are controlled by the **app_*** variables in Section 4.1.1.1, “**group_vars/all.yml**”:

- **app_template_path** is the path (relative to the root of the source repo) where the application template is stored.
- **app_name** specifies a name used for object instances resulting from the template, like the Build and Deployment configurations
- **app_base_tag** refers to the ImageStreamTag that contains the base image for the application’s build.

The example automation assumes the application template accepts at least the following parameters:

- **NAME**: suggested name for the objects generated by the template. Obtained from the **app_name** variable.
- **TAG**: generated by the pipeline as **VERSION-BUILDNUMBER**, where **VERSION** is the contents of the **app/VERSION** file and **BUILDNUMBER** is the sequential number of the build that generates the image.
- **REGISTRY**: the URL for the registry. Obtained from **central_registry_hostname**.
- **REGISTRY_PROJECT**: the namespace in the registry under which built images are kept. Obtained from the **project_name** of the registry project configuration.
- **IMAGESTREAM_TAG**: In practice this means that the application images are expected to be at:

  ```
  ${REGISTRY}/${REGISTRY_PROJECT}/${NAME}:${TAG}
  ```

  and the ImageStreamTag **${IMAGESTREAM_TAG}** points there.

Also, as the application used as an example in this reference implementation is a NodeJS based application, the pipeline includes a stage for automated testing using **npm test**.

4.2.3. Jenkins

The Jenkins instance driving the automation is deployed from a custom image which is itself built using S2I. This enables customization of the official base image through the addition of a custom list of plugins. The process is described in the Jenkins image documentation.

The Build Configuration **`jenkins-custom`** defines this S2I build for Jenkins itself. This build is also driven by Jenkins through the Section 3.3.3, "jenkins-lifecycle" pipeline, which watches for changes in the repo and triggers the pipeline-based build when appropriate.

These are the various components of this process in more detail:

- jenkins-custom-build.yaml contains the S2I build configuration.
- plugins.txt contains a list of plugins to install into the Jenkins custom image during the S2I build.
- jenkins-pipeline.yaml is a template to deploy the pipeline that manages the build of the custom Jenkins instance. The pipeline itself is defined in its own Jenkinsfile.
- jenkins-master.yaml is a template from which the deployment related objects are created: Deployment Configuration, Services, associated Route, etc. The deployment uses a Persistent Volume Claim associated to Jenkins' data storage volume (**`/var/lib/jenkins`**) so the data remains accessible upon potential container restarts and migrations across nodes.

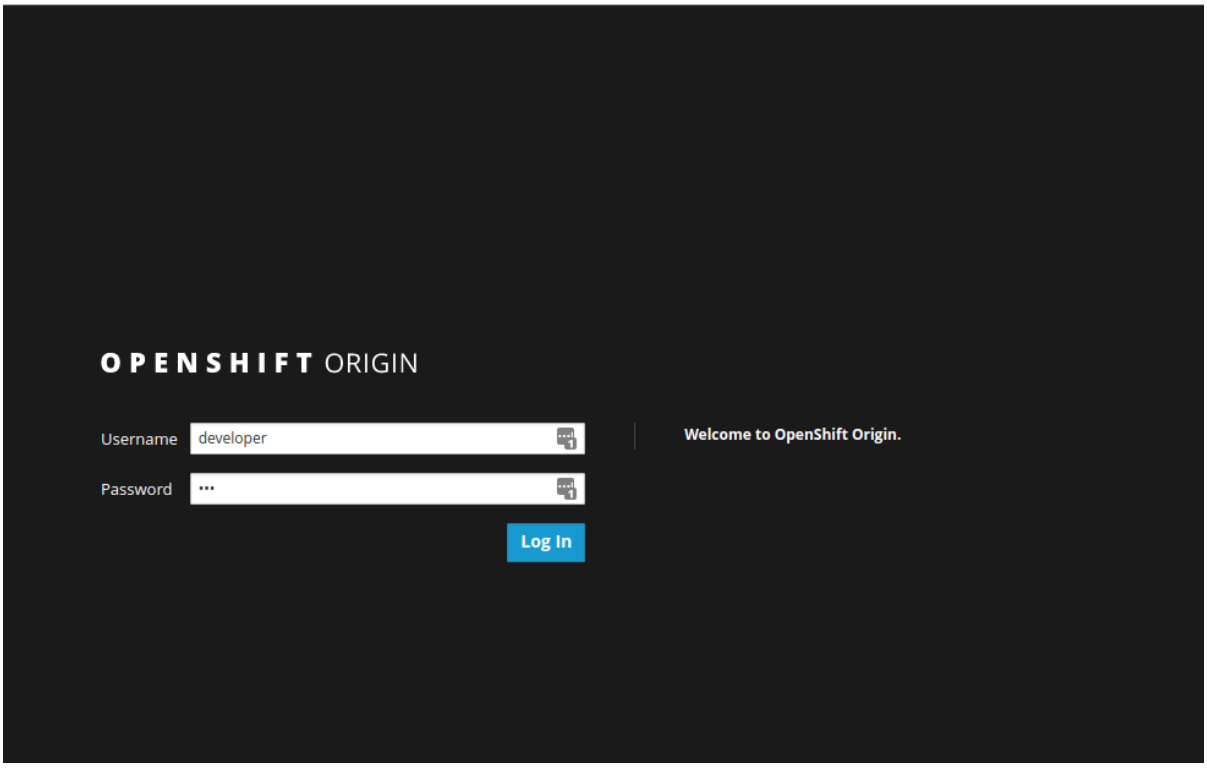
4.3. EXECUTE AND MONITOR JENKINS APP-PIPELINE

After the initial Ansible automation has run, log in to OpenShift and continue the pipeline automation.

1. Login into OpenShift

```
$ oc login
```

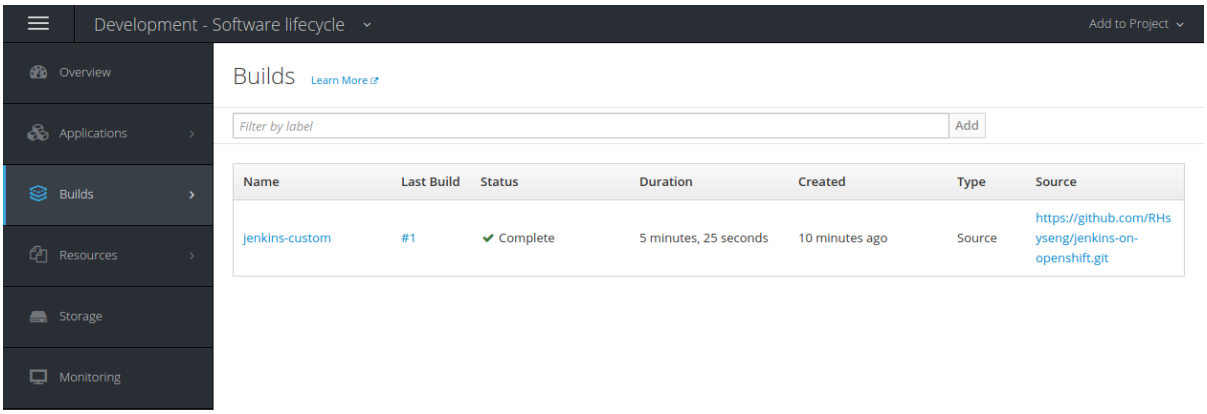
Figure 4.1. OpenShift Login



2. Jenkins is configured as a S2I build. Confirm the build completes

```
$ oc get build -l 'buildconfig=jenkins-custom' --template '{{with
index .items 0}}{{.status.phase}}{{end}}'

Complete
```

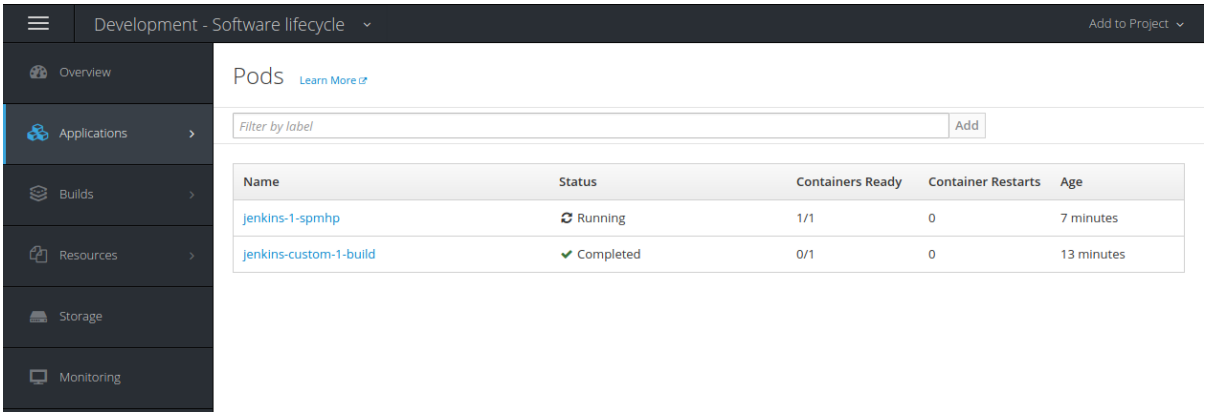
Figure 4.2. OpenShift Builds



3. Once the S2I build of Jenkins is complete, a deployment of Jenkins starts automatically. Confirm the Jenkins pod is running and the Jenkins application has started successfully by reviewing the log for **`INFO: Jenkins is fully up and running`**.

```
$ oc get pod -l 'name==jenkins'
$ oc logs -f dc/jenkins

... [OUTPUT ABBREVIATED] ...
INFO: Waiting for Jenkins to be started
Nov 28, 2017 2:59:40 PM jenkins.InitReactorRunner$1 onAttained
INFO: Loaded all jobs
... [OUTPUT ABBREVIATED] ...

Nov 28, 2017 2:59:44 PM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running

... [OUTPUT ABBREVIATED] ...
```

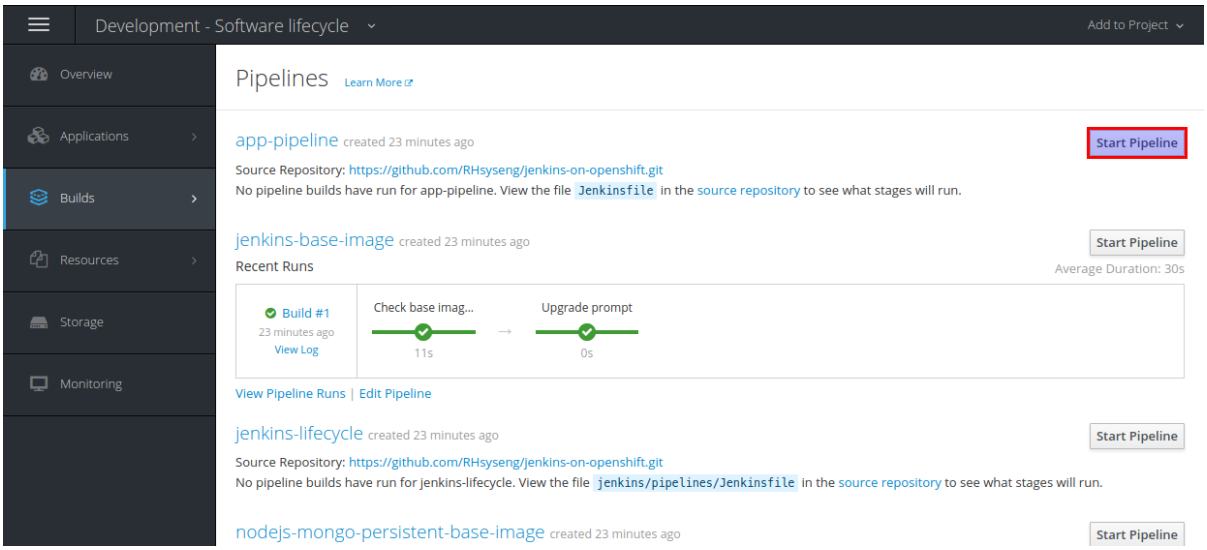
Figure 4.3. OpenShift Pods



4. Once Jenkins is up and running the application pipeline can be started. Click **[Builds]** → **[Pipeline]** to navigate to the OpenShift Pipeline view.

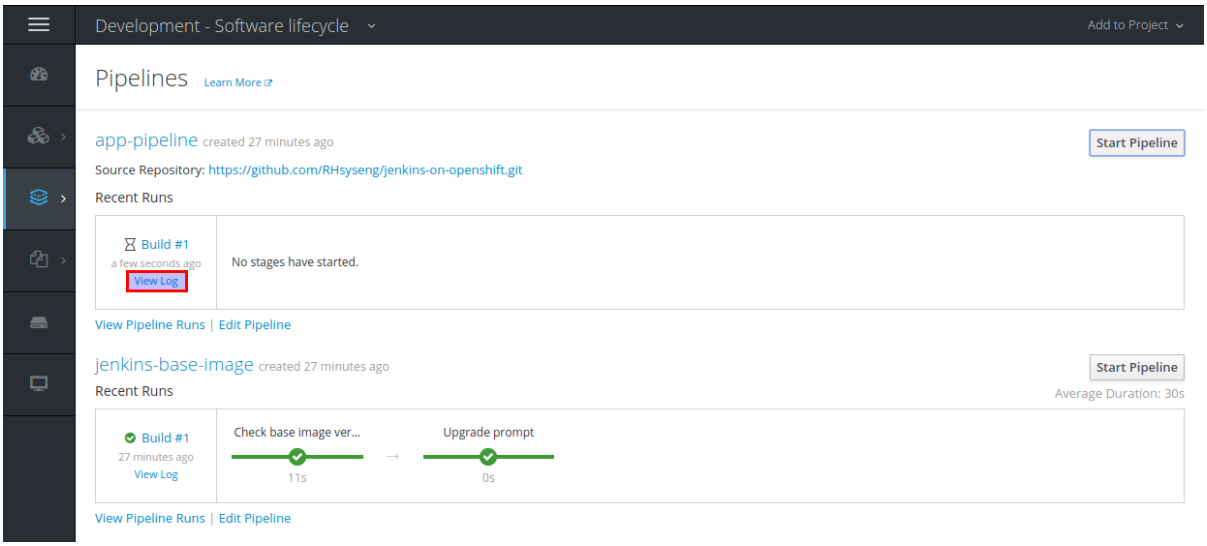
```
$ oc start-build app-pipeline

build "app-pipeline-1" started
```

Figure 4.4. OpenShift Pipeline View



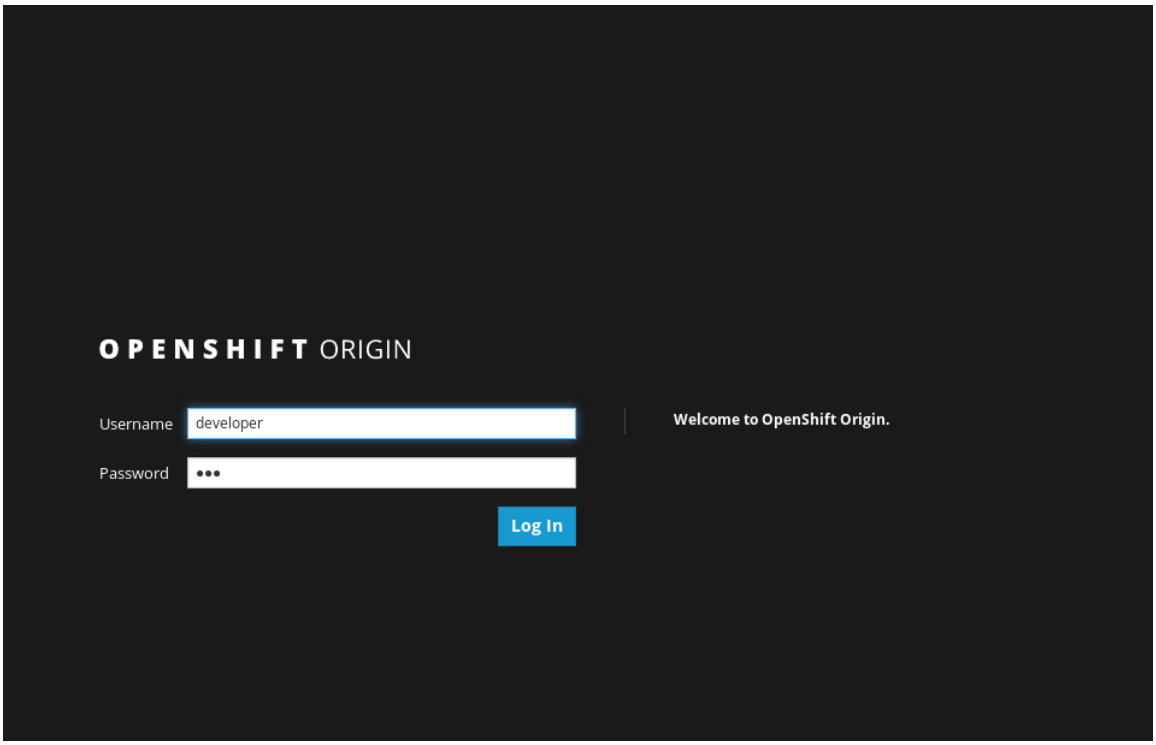
5. To get a detailed view of the pipeline progress click **[View Log]** which will launch the Jenkins console output

Figure 4.5. OpenShift Pipeline View Log



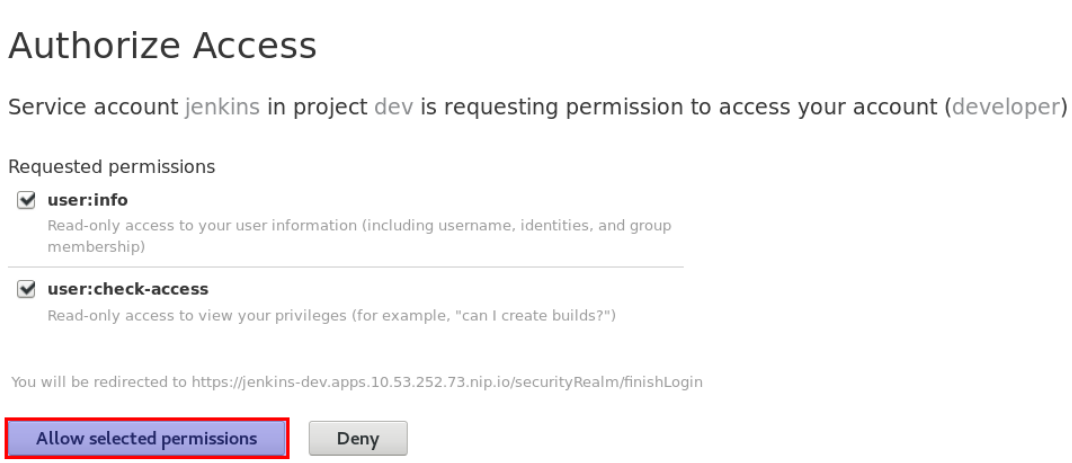
a. Upon clicking **[View Log]** you may be prompted to log in with OpenShift.

Figure 4.6. OpenShift OAuth



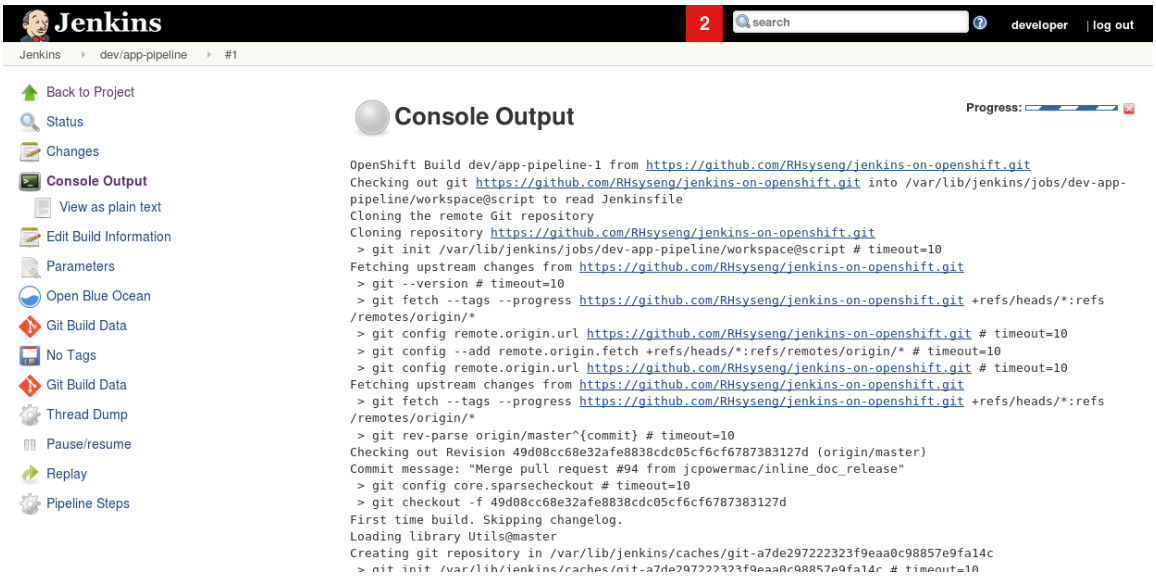
b. If this is the first time accessing the Jenkins console you will need to authorize access to Jenkins from your OpenShift account.

Figure 4.7. OpenShift OAuth permissions



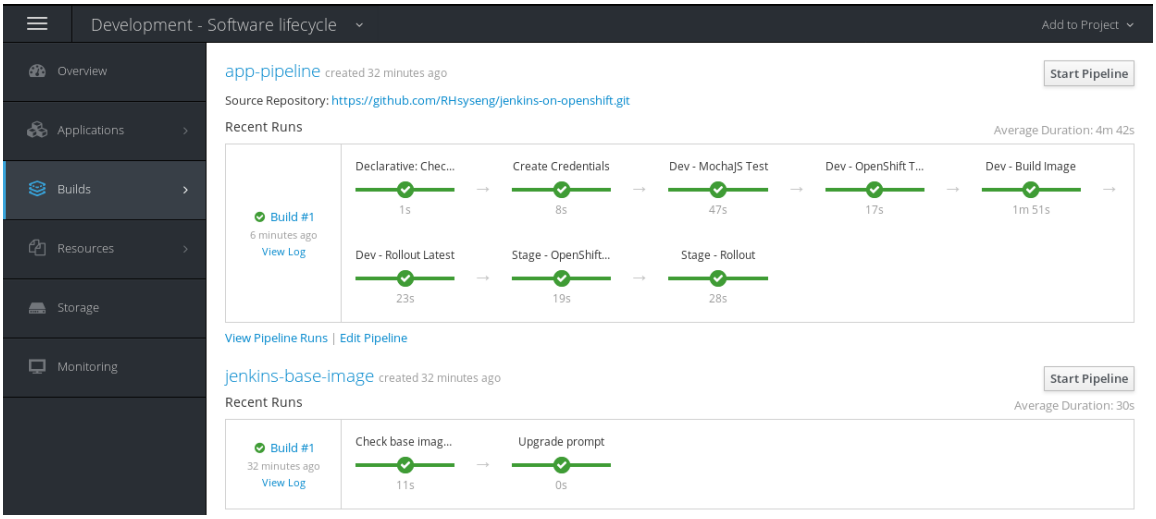
c. Below are the pipeline console logs for the **`app-pipeline`**.

Figure 4.8. Jenkins pipeline console



d. Returning to the OpenShift WebUI. The **[Builds]** → **[Pipeline]** view displays the completed pipeline and executed stages.

Figure 4.9. OpenShift Pipeline stage view

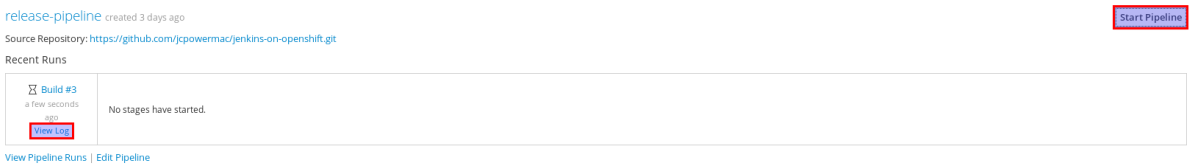


4.4. EXECUTE AND MONITOR JENKINS RELEASE-PIPELINE

1. Now that the **`app-pipeline`** has run and completed successfully promotion of the production image is possible. Return to the OpenShift WebUI pipeline view. There press **[Start Pipeline]**. To get a detailed view of the pipeline progress click **[View Log]** which will launch the Jenkins console output. Upon clicking **[View Log]** you may be prompted to login with OpenShift credentials.

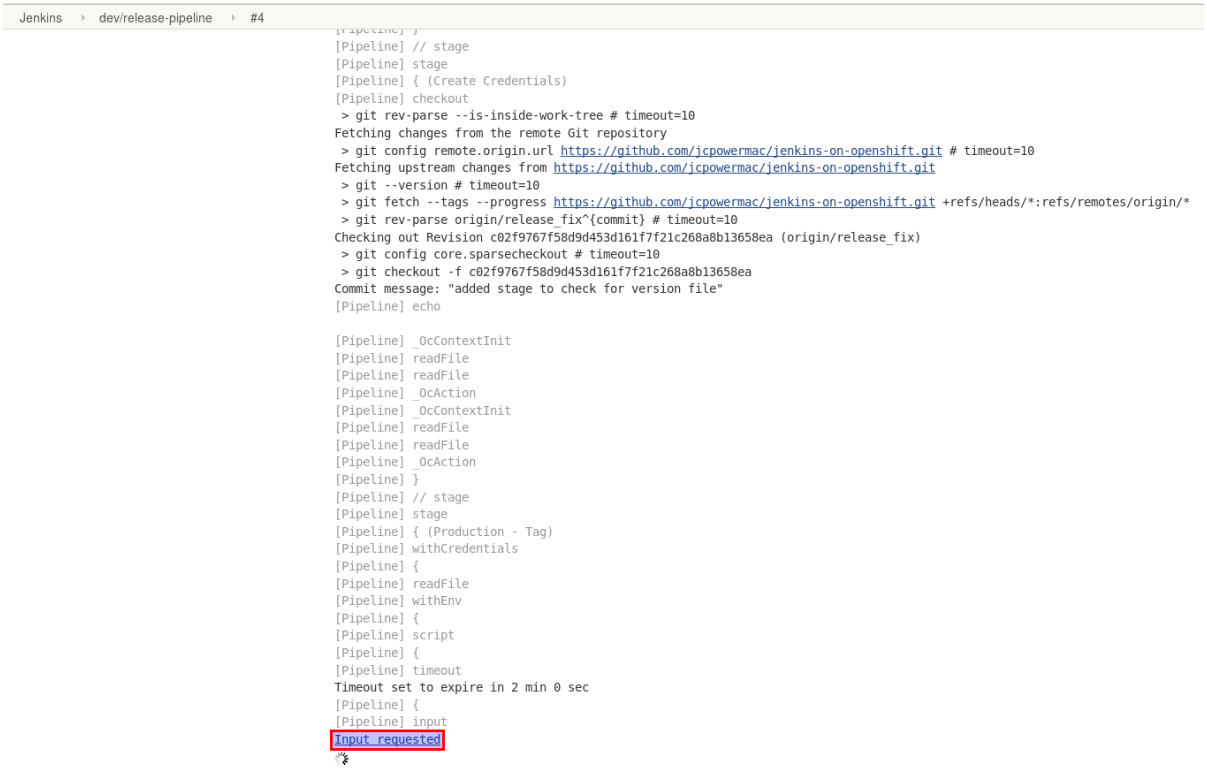
```
$ oc start-build release-pipeline

build "release-pipeline-1" started
```

Figure 4.10. OpenShift Pipeline



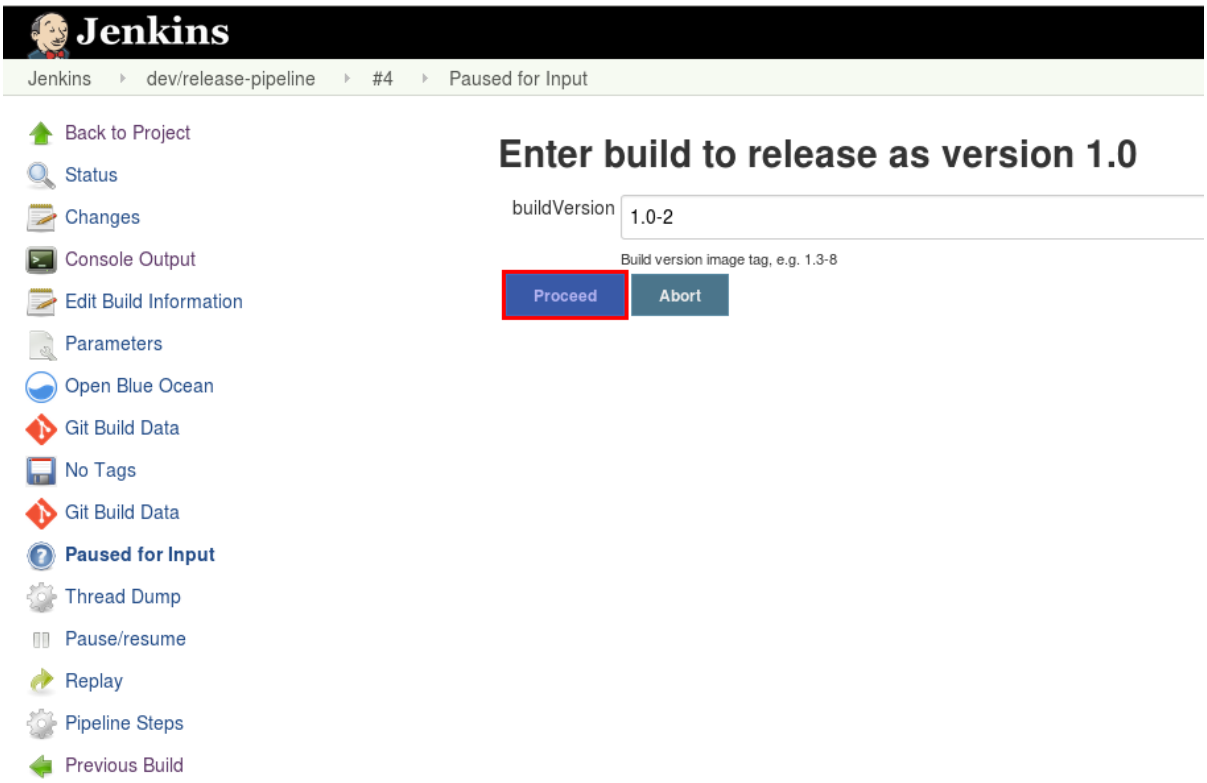
2. The pipeline will request the tag to be promoted to production. To access the input field click the **[Input requested]** link.

Figure 4.11. OpenShift Pipeline View Log



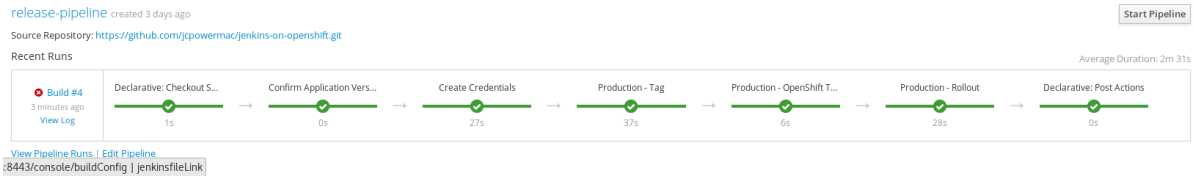
3. Enter the build tag of the image to be promoted to production. Once complete press **[Proceed]** to continue.

Figure 4.12. Jenkins input - Image tag



4. Returning to the OpenShift WebUI. The **[Builds]** → **[Pipeline]** view displays the completed pipeline and executed stages.

Figure 4.13. OpenShift Pipeline stage view



5. Log in to the production cluster and project. Click the application link which is available in the project's home.

NodeJS Example Application

Welcome to your Node.js application on OpenShift

How to use this example application

For instructions on how to use this application with OpenShift, start by reading the Developer Guide.

Deploying code changes

The source code for this application is available to be forked from the OpenShift GitHub repository. You can configure a webhook in your repository to make OpenShift automatically start a build whenever you push your code:

1. From the Web Console homepage, navigate to your project
2. Click on Browse > Builds
3. Click the link with your BuildConfig name
4. Click the Configuration tab
5. Click the "Copy to clipboard" icon to the right of the "GitHub webhook URL" field
6. Navigate to your repository on GitHub and click on repository settings > webhooks > Add webhook
7. Paste your webhook URL provided by OpenShift
8. Leave the defaults for the remaining fields — that's it!

After you save your webhook, if you refresh your settings page you can see the status of the ping that Github sent to OpenShift to verify it can reach the server.

Note: adding a webhook requires your OpenShift server to be reachable from GitHub.

Working in your local Git repository

If you forked the application from the OpenShift GitHub example, you'll need to manually clone the repository to your local system. Copy the application's source code Git URL and then run:

```
$ git clone <git_url> <directory_to_create>

# Within your project directory
# Commit your changes and push to OpenShift

$ git commit -a -m 'Some commit message'
$ git push
```

Managing your application

Documentation on how to manage your application from the Web Console or Command Line is available at the Developer Guide.

Web Console

You can use the Web Console to view the state of your application components and launch new builds.

Command Line

With the OpenShift command line interface (CLI), you can create applications and manage projects from a terminal.

Development Resources

- OpenShift Documentation
- Openshift Origin GitHub
- Source To Image GitHub
- Getting Started with Node.js on OpenShift
- Stack Overflow questions for OpenShift
- Git documentation

Request information

Page view count: 1

DB Connection Info:

Type: MongoDB

URL: mongodb://172.30.188.236:27017/sampledb

CHAPTER 5. CONCLUSION

The integration of Jenkins with OpenShift addresses many challenges teams face when starting to implement Application CI/CD.

This reference implementation covered the following topics:

- Application image promotion example between clusters and projects
- Architectural overview
- Deployed an example Application CI/CD using Ansible, Jenkins, and OpenShift.
- Provided guidance with developing automation.

For any questions or concerns, please email refarch-feedback@redhat.com and ensure to visit the Red Hat Reference Architecture page to find about all of our Red Hat solution offerings.

APPENDIX A. CONTRIBUTORS

Aaron Weitekamp, content provider

Christoph Görn, content reviewer

Joseph Callen, content provider

Pep Turró Mauri, content provider

Rayford Johnson, content reviewer

Tommy Hughes, content reviewer

Andrew Block, content reviewer

Gabe Montero, content reviewer

Erik Jacobs, content reviewer

Ben Parees, content reviewer

Justin Pierce, content reviewer

Scott Collier, content reviewer

APPENDIX B. KNOWN ISSUES

- The **`app-pipeline`** and **`release-pipeline`** may fail in the DeploymentConfig rollout stage if the OpenShift template is modified after initial deployment. As of this writing there is an open OpenShift issue BZ 1515902 to resolve this issue.
- Modification of the Jenkins BuildConfig and an additional ImageStream were required in OpenShift Container platform v3.7. This was due to the **`openshift`** project's **`jenkins`** ImageStream using the latest tag instead of v3.7. As of this writing there is an open OpenShift issue BZ 1525659 to resolve this issue.
- At the time of this writing if the OpenShift v3.7 client is used with the provided Ansible playbooks push and pull of the registry will fail with **`unauthorized: authentication required`**. A workaround has been implemented in Ansible to download and use the upstream origin v3.6.1 client. For further information see BZ 1531511, BZ 1476330, or PR 18062.

APPENDIX C. OPENSHIFT TEMPLATE MODIFICATIONS

An existing OpenShift example application **nodejs-ex** was used for this project. It provides an OpenShift Template requiring modification to support the image promotion and deployment process. Below is an annotated list of modified sections.

ImageStream

```
{
  "kind": "ImageStream",
  "apiVersion": "v1",
  "metadata": {
    "name": "${NAME}",
    "annotations": {
      "description": "Keeps track of changes in the application image",
      "openshift.io/image.insecureRepository": "true"
    }
  },
  "spec": {
    "tags": [
        {
            "from": {
                "kind": "DockerImage",
1               "name": "${REGISTRY}/${REGISTRY_PROJECT}/${NAME}:${TAG}"
2           },
            "name": "${IMAGESTREAM_TAG}",
3           "importPolicy": {
                "insecure": true,
                "scheduled": true
4           }
        }

    ]
  }
},
```

1. Since an external registry is used the tags kind will be **DockerImage**.
2. **name** points to the url of the Docker image including the specific tag that will be imported.
3. This is the tag of the ImageStream within the project that will be deployed.
4. Keep the image up to do date based on any changes in the source registry.

BuildConfig

```
{
  "kind": "BuildConfig",
  "apiVersion": "v1",
  "metadata": {
```

```
    "name": "${NAME}",
    "annotations": {
      "description": "Defines how to build the application",
      "template.alpha.openshift.io/wait-for-ready": "true"
    }
  },
  "spec": {
    "source": {
      "type": "Git",
      "git": {
        "uri": "${SOURCE_REPOSITORY_URL}",
        "ref": "${SOURCE_REPOSITORY_REF}"
      },
      "contextDir": "${CONTEXT_DIR}"
    },
    "strategy": {
      "type": "Source",
      "sourceStrategy": {
        "from": {
          "kind": "ImageStreamTag",
          "namespace": "${NAMESPACE}",
          "name": "nodejs:6"
        },
        "env":  [
          {
            "name": "NPM_MIRROR",
            "value": "${NPM_MIRROR}"
          }
        ]
      }
    },
    "output": {
      "to": {
        "kind": "DockerImage",                                          (1)
        "name": "${REGISTRY}/${REGISTRY_PROJECT}/${NAME}:${TAG}"        (2)
      }
    },
    "triggers": [],                                                     (3)
    "postCommit": {
      "script": "npm test"
    }
  }
},
```

1. Changed from ImageStreamTag to DockerImage so an external registry can be utilized.
2. Path to the Docker Registry the image will be pushed after the build process is complete.
3. All the triggers for the BuildConfig have been removed. The initiation of the build process will be handled by Jenkins.

DeploymentConfig

```
{
  "kind": "DeploymentConfig",
```

```
  "apiVersion": "v1",
  "metadata": {
    "name": "${NAME}",
    "annotations": {
      "description": "Defines how to deploy the application server",
      "template.alpha.openshift.io/wait-for-ready": "true"
    }
  },
  "spec": {
    "strategy": {
      "type": "Recreate"
    },
    "triggers": [                                          (1)
      {
        "type": "ImageChange",
        "imageChangeParams": {
          "automatic": false,                              (2)
          "containerNames": [
            "nodejs-mongo-persistent"
          ],
          "from": {
            "kind": "ImageStreamTag",
            "name": "${NAME}:${IMAGESTREAM_TAG}"           (3)
          }
        }
      }
    ],
... [OUTPUT ABBREVIATED] ...
```

1. Removed ConfigChange trigger. The DeploymentConfig will be rolled out using Jenkins.
2. Changed automatic to false. Jenkins will roll out new versions through the pipelines.
3. Parameterized the **IMAGESTREAM_TAG**

Parameters

```
... [OUTPUT ABBREVIATED] ...
{
  "name": "TAG",                                           (1)
  "displayName": "Current Docker image tag in registry",
  "description": "",
  "required": true
},
{
  "name": "IMAGESTREAM_TAG",                               (2)
  "displayName": "Current ImageStreamTag in OpenShift",
  "description": "",
  "required": true
},
{
  "name": "REGISTRY",                                      (3)
  "displayName": "Registry",
  "description": "The URL of the registry where the image resides",
```

```
    "required": true
  },
  {
    "name": "REGISTRY_PROJECT",                    4
    "displayName": "The registry project where the image resides.",
    "description": "",
    "required": true
  },
  ... [OUTPUT ABBREVIATED] ...
```

1. Required to control the deployment of certain tagged versions of an image
2. Required to control the deployment of certain tagged versions of an image
3. Required to use an external registry
4. Required to use an external registry

APPENDIX D. REVISION HISTORY

Revision 3.7.0-0 **Januaray 31, 2018** **aweiteka jcallen jturro**

- **Initial release**