TABLE OF CONTENTS

CHAPTER 1: WHAT IS CI/CD?

Continuous Integration (CI) and Continuous Delivery (CD) are two of many best practices put to use in devops teams.

Initially just a collection of practices, they have evolved into much more than that: a set of operating principles, a coding philosophy, a culture.

Together, they form the baseline for another best practice called Continuous Release (CR), also called by some Continuous Deployment (CDE), which is often the ultimate step companies are striving towards in the area of pushing software updates quickly and painlessly to their users.

Although their names are similar, they address very different aspects of software development and lifecycle though.

Section 1.1: Continuous Integration

Continuous integration, in the simplest terms, means that a team of developers working on a software project should coordinate the results of their work as frequently as possible, preferably many times a day. This seemingly simple requirement has a number of important consequences though, not least elusive of which is the correct sizing of tasks everyone is working on.

Imagine a team working on a web application - there is someone developing the frontend, someone in charge of the database integration, someone working on the web services exposed by the application, etc.

If they were to agree completely upon everything they have to implement from the start out, there would only be one integration point in the project: when everyone finishes their part of the job. It would be a big and painful integration point, however, for several reasons you can probably imagine quite easily:

- some team members would take more time than others to finish, which would inevitably delay the integration point until the very last team member completed their work;
- in turn, that would prevent those that are already finished from seeing potential flaws in their work early and start working towards fixing them, rather than wait for everyone to finish and only then face a host of improvements that are needed;
- some team members would understand the instructions differently from others, resulting in incompatibilities between their implementations, which would only ever be noticed at the (seeming) end of the project, showing up as hundreds of additional items on everyone's schedule and inevitable "whose fault is it" inquisition;
- for teams that have several members working on the same module (like business logic components, for example), work would be next to impossible without either many code conflicts or regular code merges (which in themselves are mini integration points), resulting in isolated module evolution due to challenges encountered in those merges, and thus even further drift away from the rest of the application;
- last but not least, the software project workflow would consist of initial implementation first, and hundreds of bug tickets later, which translates to a very boring work pattern nobody likes.

In a typical agile project, tasks are broken down into much smaller units of work which are more easily completed within a short time-frame, allowing team members to integrate their work with the whole project up to several times a day, producing a number of benefits to the project, such as:

- smaller tasks are more easily reassignable;
- their outcomes are generally easier to test in an automated way;
- since integrations can occur several times per day, bugs and incompatibilities are found earlier, and can be acted upon quicker;
- each producing a tangible result, small tasks greatly contribute to transparency of the project's progress;
- this transparency allows everyone to be up-to-date with what shape the application is taking, and steer the outcome more proactively even during development;
- upon failure, smaller tasks tend to produce less damage to the overall project, in the form of delays and/or destabilizing the codebase;

It is obvious that this approach has some clear requirements if we want it to work reliably:

- developers must agree to use a source code management (SCM) repository - their source code is regularly merged into common code-base in that repository;
- there must be sufficiently extensive unit tests for each individual component developed (this helps prove that a single component is working according to specification) - some teams decide to use test-driven design (TDD) to fulfill this requirement;
- there must be very basic, but representative integration tests that exercise logical functions of several modules in conjunction, such as making a purchase - these are usually performed at the lowest possible layer (such as simply invoking business logic components internally to the application, and by mocking any external dependencies such as databases and other web services).

The above three conditions make it possible to test the correct behavior of application components in an automated way by pulling the code out of the SCM repository immediately after an integration point, running the unit tests first, and then executing the integration tests.

Any failures would be automatically reported to the team, prompting them to start working on eliminating them prior to implementing any additional new features. Every subsequent code merge would reiterate this process until all failures are eliminated.

Some teams may implement additional guidelines to help ensure everybody adheres to the good practices, such as having:

- a static code analysis tool to test against known anti-patterns, but more importantly, measure the portion of code tested in the unit and integration tests, with a minimum percentage requirement to pass the integration cycle;
- a mandatory code commit interval, aimed at preventing developers from stalling the commits because “it's not ready yet”;

- strict formal requirements for the ability to create branches in the code, to prevent developers from creating them just so that they can work “without being disturbed by other people's changes”;

All of the above guidelines help ensure that the source code for the entire application is as up to date as possible, easy to maintain as a whole, and remains in a healthy overall state.

Section 1.2: Continuous Delivery

If continuous integration is predominantly a team-development-oriented effort, continuous delivery is a much wider term, as it refers to automation of every conceivable requirement for the deployment of the new version of the application, with the exception of production deployment itself.

To put it as simply as possible, a continuous delivery iteration goes through a set of validations that are necessary to prove that a software product is going to be deployed successfully after any given change. This may include steps such as:

- isolating any configuration changes that need to be performed during an application rollout and automating the process of applying them (and rolling them back);
- verifying that the latest release candidate works correctly with existing versions of dependencies such as database schema, downstream web service APIs, and other integration points, by running a series of automated integration tests in an environment mimicking the production as closely as possible;
- making sure application components cooperate correctly by running a set of automated acceptance tests (usually at different layers, such as externally invoking a web service as if a web browser would, and then performing the equivalent workflow through the web user interface - this makes it easier to pin-point where exactly a failure occurred);
- testing the quality of application components against a list of known vulnerable modules;
- implementing configurable feature toggles for features that are not ready for prime time yet.

In other words, the term *continuous* in continuous delivery refers to the effort of continuously ensuring the application is successfully deployable at any given time in any given stage of development.

A continuous delivery implementation will typically incorporate all the steps involved in continuous integration, and include several additional series of tests, where each of the above requirements is verified.

These tests are usually performed across a series of different execution environments, commonly known as *test/int/QA/UAT* stages, where the simplicity of an initial test environment (for the sake of agility in setting it up, and lower cost of operation) is gradually reduced in favor of greater resemblance to production environment (for the sake of representability of the tests).

NOTE: While obvious requirements for a successful deployment to production: the choice of deployment process, methods used to perform a seamless roll-out, and measures implemented to ensure a safe roll-back at any time it may be required, identifying any potential point-of-no-return stages, are not necessarily the concern of a continuous delivery process.

In CD, releasing an application to production is a business decision. The CD process is in place to ensure we always have solid, reliable, and releasable artifacts at our disposal. The business decides whether to do production releases on a nightly, weekly, bi-weekly, some other schedule, or even on-demand, possibly even after interactive user acceptance testers have signed the application off for a production release.

Section 1.3: Continuous Release

Also called “Continuous Deployment” by some, continuous release is the process of carefully replacing manual deployment steps following a production sign-off, with automated ones, such that a number of service level requirements is maintained.

Instead of someone (or a team of people) coordinating each step of the application deployment to production environment, these steps end up being implemented in an automated way.

Some strategies employed in CR may include:

- blue/green production environments, where post-deployment smoke tests are performed in the inactive environment prior to roll-over;
- A/B testing scenarios where multiple different versions of an application may be running at the same time, and only a small portion of the customers is exposed to the new version for a while, allowing for quick rollback and minimal damage in case of problems;
- different deployment strategies (such as rolling deployments), where the exact steps of a rollout depend on the nature of the application, and even possible data structure changes, either of which might prevent us from being able to run more than one version (or even more than one instance) of the application at the same time;
- composite applications where each of the UI components (such as the catalog, item preview, suggestions, etc.) is served by a different backend module, allowing for independent upgrades of individual modules;
- going all the way back to how modules are coded, the use of microservice fault tolerance development patterns such as circuit breakers and fallback responses in case a dependency becomes suddenly unavailable;
- clear and precise smoke tests, along with reliable rollback procedures for every step of a rollout;
- ongoing performance and functional monitoring for every application module, and every logical function.

It is important to remember that, while CR is a wonderful practice if implemented, it may not be suitable for everything. There are applications, particularly legacy monolithic ones, but also others, that can not easily be modularized, can not easily scale horizontally, thus providing a poor foundation for a typical rolling deployment with zero downtime.

There are even use-cases where implementing CR is more trouble than it is worth - in a typical offline back-office environment, the business operates on a very predictable schedule, giving the operations team plenty of opportunity for scheduled outages, rather than risking the deployment of a new version during working hours, potentially disrupting hundreds or thousands of customers.

However, even in such environments, having some (or most) of the building blocks for CR in place can immensely simplify and speed up the manual deployment process, and make it more reliable.

Section 1.4: Implementing CI/CD/CR

As you can probably conclude from the above descriptions of the continuous practices, implementing "the three Cs" is primarily an organizational concern involving a considerable amount of pre-meditation, a lot of policies, rules, and procedures to be in place, and ongoing adjustments and improvements to be made to all the processes every time a potential problem is encountered.

No application can do that for us. In as much as they can, however, certain tools are of immense value in implementing all of the above (some, like SCM, are even a requirement). Most of these tools provide users with some sort of a build environment, on-premise or cloud-based, which can be configured to run more or less complex build jobs, consisting of various tests. They will also typically feature a plug-in framework which can be used to further integrate these tools with our infrastructure, or add functionality to a simple build job.

Two of the most popular on-premise tools are Jenkins and Hudson, whereas most cloud-based SCM services like GitHub, GitLab, BitBucket and similar, offer their own CI/CD frameworks. Alternatively, there are third-party providers which allow you to integrate their cloud-based solution with your own infrastructure.

In the rest of this module we will be looking at Jenkins as an example of an open-source tool that provides an excellent framework for the implementation of all three continuous practices.

Quiz: CI/CD/CR Concepts

Intense Quote List

- List item 1
- List item 2

Quote List

- List item 1
- List item 2

Non-indexed Subtitle

Another bit of text.

CHAPTER 2: JENKINS AND OPENSHIFT

Jenkins was conceived in 2004 under the name of Hudson, by Kohsuge Kawaguchi, then a Sun Microsystems employee, for the simple reason that he was usually the one developer whose code caused the integrated builds to fail, and he wanted to have a program that would catch these failures before his colleagues did. The rest, as they say, is history.

Section 2.1: Jenkins as a CI/CD Server

At its simplest, you can think of Jenkins as a build server. It has a web console, and a CLI, both offering the capacity to create projects, and define one or more build jobs within those projects. Those jobs can be executed manually, but also scheduled, or triggered by an external action.

Jenkins and CI

You can start using Jenkins as a simple CI server in this way - every time code is committed to the SCM server, Jenkins can start a new build and report the results to a team of people. Builds can be configured for any number of languages and build tools, although there is “first class citizen” support for more popular platforms like Java and .NET, and build tools like Maven.

The build framework of Jenkins is very customizable. For example, builds can be performed by the Jenkins service internally, or through a remote agent, which allows you to have a pool of build servers, even running different operating systems and architectures, and a master Jenkins server which controls them.

Building blocks for CD

The brilliant part is that those jobs do not actually have to build anything.

Of course, there will be a job in your project that tests if the code still builds after the merge, producing as it may, a deployable package which can also be downloaded from Jenkins whenever needed, or uploaded to Nexus or Artifactory as a build artifact used by other builds or deployments.

But there can also be a build job that runs the automated unit tests and reports the results in case of failures, or just stores the test reports on the server for subsequent review. Or there can be a generic build job that runs an arbitrary tool your company uses to perform static code analysis. Or, and this is where it gets really interesting, a job can invoke an external service to perform an action on its behalf, such as start an OpenShift build or push a container image to a registry. This allows us to define most, if not all, of the steps in a CD workflow within Jenkins.

Initially, the concept of chained builds was used to link different steps together. Because chained builds were very popular, but difficult to maintain (as the upstream build outcome had to be configured as a trigger for the next build in the downstream build’s configuration and this led to extremely poor visibility), several alternative solutions were developed, among which a simple delivery pipeline plug-in (which did the job of managing upstream/downstream build configurations for you), a more general build flow plug-in (which was capable of declaring a pipeline as a series of builds and invoke them itself), and ultimately, a simple job DSL (domain-specific language) that allowed one to express the inputs to a series of events (such as an SCM URL), triggers (such as a schedule, or a commit event), and the steps that the job should take in the course of its execution.

The single remaining issue that led to further development in the area of "freestyle jobs" as they were called, was that as they were, these jobs either still relied on other job type definitions and just orchestrated them, or were extremely low-level and platform-dependent in any other action they were to execute. For example, one freestyle job could invoke a BuildJob, a TestJob, and a DeployJob, but those three still had to be configured as separate Jenkins jobs. It could also copy a file somewhere, but that had to be an inline shell script defined and invoked in the job definition.

Release/Deployment Pipelines

Enter pipelines. With pipelines came the concept of stages, which are series of related steps grouped together (much like a BuildJob vs. a TestJob would be), but defined in the same pipeline, not across several different resources in a project. And instead of invoking a pre-defined job rather statically, or performing a low-level system operation, pipeline plug-ins expose simple DSL operations one can use to execute more complex operations, for example start a remote build, send an email or publish an MQTT message somewhere, by writing as little as a single pipeline instruction.

Through the use of a simple, but powerful, plug-in architecture, Jenkins has gained capability to talk to different SCM services, perform builds in different environments understanding different types of build systems, interface with all sorts of external systems, and most recently, talk to Kubernetes and OpenShift as well.

Section 2.2: Deploying Jenkins in OpenShift

There are two general ways of making Jenkins work with OpenShift. It is possible to deploy a stand-alone master Jenkins in a traditional way and have it create agent pods in OpenShift as needed, or deploy it as a pod itself, and have it orchestrate pipelines from within an OpenShift project.

While the first one may be a good choice for when an existing build infrastructure is already in place, it requires a number of additional plug-ins to be installed, and configuration adjustments to be made both on Jenkins and OpenShift sides.

We will therefore focus on the second one, as it is much simpler and more integrated.

Jenkins ImageStreams and Templates

Out of the box, there are a number of `ImageStream`s created in the `openshift` project of OCP installation, and among them, there is also one that contains several Jenkins images, ready to be deployed to any project:

```
[student@workstation ~]$ oc -n openshift get is jenkins
NAME      DOCKER REPO                                        TAGS          UPDATED
jenkins   docker-registry.default.svc:5000/openshift/jenkins   2,latest,1    5 days ago

[student@workstation ~]$ oc -n openshift describe is jenkins
Name:              jenkins
Namespace:         openshift
Created:           2 months ago
Labels:            <none>
Annotations:       openshift.io/display-name=Jenkins
                   openshift.io/image.dockerRepositoryCheck=2019-01-25T10:41:28Z
Docker Pull Spec:  docker-registry.default.svc:5000/openshift/jenkins
Image Lookup:      local=false
```

```
Unique Images:    2
Tags:             3

1
  tagged from services.lab.example.com/openshift3/jenkins-1-rhel7:latest
...
(content omitted)
...
2 (latest)
  tagged from services.lab.example.com/openshift3/jenkins-2-rhel7:latest
...
(content omitted)
```

There are also two templates available to simplify the deployment of these images, one without, and one with persistent storage:

```
[student@workstation ~]$ oc -n openshift get templates | grep jenkins
jenkins-ephemeral   Jenkins service, without persistent storage....  7 (all set)    6
jenkins-persistent  Jenkins service, with persistent storage....     9 (all set)    7

[student@workstation ~]$ oc -n openshift process jenkins-persistent --parameters
NAME                            DESCRIPTION                    GENERATOR    VALUE
JENKINS_SERVICE_NAME            The name of the OpenShift...                jenkins
JNLP_SERVICE_NAME               The name of the service...                  jenkins-jnlp
ENABLE_OAUTH                    Whether to enable OAuth...                  true
MEMORY_LIMIT                    Maximum amount of memory...                 512Mi
VOLUME_CAPACITY                 Volume space available...                   1Gi
NAMESPACE                       The OpenShift Namespace...                  openshift
DISABLE_ADMINISTRATIVE_MONITORS Whether to perform memory...                false
JENKINS_IMAGE_STREAM_TAG        Name of the ImageStreamTag...               jenkins:2
ENABLE_FATAL_ERROR_LOG_FILE     When a fatal error occurs,...               false

[student@workstation ~]$ oc -n openshift describe template jenkins-persistent | tail -n8
Objects:
    Route                     ${JENKINS_SERVICE_NAME}
    PersistentVolumeClaim     ${JENKINS_SERVICE_NAME}
    DeploymentConfig          ${JENKINS_SERVICE_NAME}
    ServiceAccount            ${JENKINS_SERVICE_NAME}
    RoleBinding               ${JENKINS_SERVICE_NAME}_edit
    Service                   ${JNLP_SERVICE_NAME}
    Service                   ${JENKINS_SERVICE_NAME}
```

To deploy Jenkins, one of the options is certainly using the `oc new-app` command with either `--image-stream` or `--template` options to select the desired resource that should become the source of Jenkins' deployment. When using a template, some of the common parameters include:

Parameter Name	**Default Value**	**Explanation**
ENABLE_OAUTH	true	If true, use whatever authentication mechanism OpenShift is configured to

Parameter Name	Default Value	Explanation
		use. If false, create a static user called `admin` with the password of `password`.
`MEMORY_LIMIT`	`512Mi`	Maximum amount of memory the Jenkins container is allowed to consume. Default is good enough for builds running in agent pods, but if you are executing many builds in the Jenkins container, increase this.
`VOLUME_CAPACITY`	`1Gi`	Only in the `jenkins-persistent` template - the amount of disk space to ask for when creating a `PersistentVolumeClaim`, used primarily for build logs. Increase according to the number and frequency of your builds.
`JENKINS_SERVICE_NAME`	`jenkins`	The name not just for the K8S `Service`, but practically all the resources created by the template. Should probably not be changed.

Deploying Jenkins Automatically

Deploying Jenkins is even easier than using one of the above templates though.

Whenever a `BuildConfig` resource with the strategy of `JenkinsPipeline` is created in a project, a Jenkins service is deployed automatically from the `jenkins-ephemeral` template in the `openshift` project (unless the service already exists, of course):

```
[student@workstation ~]$ oc get all                          (1)
No resources found.

[student@workstation ~]$ cat simple-pipeline.yml
apiVersion: v1
kind: BuildConfig
metadata:
  name: simple-pipeline
spec:
  runPolicy: Serial
  source:
    type: None
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        node('maven') {
          stage('build') {
            openshiftBuild(buildConfig: 'simple', showBuildLogs: 'true')
          }
          stage('deploy') {
            openshiftDeploy(deploymentConfig: 'simple')
          }
        }
    type: JenkinsPipeline
```

```
[student@workstation ~]$ oc create -f simple-pipeline.yml    (2)
buildconfig.build.openshift.io/simple-pipeline created

[student@workstation ~]$ oc get all                          (3)
NAME                     READY     STATUS    RESTARTS   AGE
pod/jenkins-1-7dc7s      0/1       Running   0          29s
pod/jenkins-1-deploy     1/1       Running   0          33s

NAME                                DESIRED   CURRENT   READY     AGE
replicationcontroller/jenkins-1     1         1         0         33s

NAME                     TYPE         CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE
service/jenkins          ClusterIP    172.30.204.116   <none>        80/TCP      33s
service/jenkins-jnlp     ClusterIP    172.30.130.153   <none>        50000/TCP   33s

NAME                                          REVISION DESIRED CURRENT TRIGGERED BY
deploymentconfig.apps.openshift.io/jenkins 1        1       1     config,image(jenkins:2)

NAME                                            TYPE               FROM      LATEST
buildconfig.build.openshift.io/simple-pipeline   JenkinsPipeline              0

NAME                                 HOST/PORT                          PATH    SERVICES ...
route.route.openshift.io/jenkins jenkins-test.apps.lab.example.com          jenkins  ...
PORT      TERMINATION     WILDCARD
<all>     edge/Redirect   None
```

1. The project is empty from the start-out.
2. The `oc create` command reports only creating a `BuildConfig`.
3. After reviewing project content though, in addition to the `BuildConfig`, we also find those from the `jenkins-ephemeral` template, present in the project (see above for a list of resources in the persistent template).

In the case that the Jenkins service is deployed automatically, any configuration adjustments to it must be made by changing the environment variables of the deployment configuration after the automatic creation of resources has taken place, or alternatively, by modifying the default `jenkins-ephemeral` template in the `openshift` project beforehand. In the latter case, the change will affect all automatic Jenkins deployments throughout the cluster from that point in time onwards.

Additional Requirements for Jenkins Deployments

As mentioned previously, it is possible to run Jenkins jobs internally in Jenkins, as one of the master executor jobs. To provide for better scalability and isolation though, it is common to run a job by giving it to a Jenkins agent to execute.

There are two types of Jenkins agents available in OpenShift - a *Maven* and a *NodeJS* agent. If your pipelines need to execute certain steps through those agents, Jenkins will also need to know where to pull the agent images from. Luckily, there is an easy way of telling that to the Jenkins pod - just by setting two environment variables in the `DeploymentConfig`:

- `MAVEN_SLAVE_IMAGE` is the image URL for the Maven agent

- `NODEJS_SLAVE_IMAGE` should point to the URL for the NodeJS agent image

For example, to have the Jenkins pod create slave pods from images stored in the `services.lab.example.com` registry, you would do the following:

```
[student@workstation ~]$ oc set env dc/jenkins --list
# deploymentconfigs/jenkins, container jenkins
OPENSHIFT_ENABLE_OAUTH=true
OPENSHIFT_ENABLE_REDIRECT_PROMPT=true
DISABLE_ADMINISTRATIVE_MONITORS=false
KUBERNETES_MASTER=https://kubernetes.default:443
KUBERNETES_TRUST_CERTIFICATES=true
JENKINS_SERVICE_NAME=jenkins
JNLP_SERVICE_NAME=jenkins-jnlp

[student@workstation ~]$ oc set env dc/jenkins \
> MAVEN_SLAVE_IMAGE=services.lab.example.com/openshift3/jenkins-agent-maven-35-\
> rhel7:latest \
> NODEJS_SLAVE_IMAGE=services.lab.example.com/openshift3/jenkins-agent-nodejs-8-\
> rhel7:latest
deploymentconfig.apps.openshift.io/jenkins updated
```

The second command above would re-deploy the Jenkins pod using the new settings.

Another important consideration is when Jenkins is being used to orchestrate events across multiple projects. The Jenkins pod must have permission to access all those projects and perform operations within them (such as start builds or tag images). For that purpose, the Jenkins templates always create a `ServiceAccount` resource for the Jenkins pod to run as, called `jenkins`.

When configuring Jenkins to run cross-project pipelines, we need to make sure that this service account has been assigned a sufficiently privileged role in all the remote projects (usually that role is `edit`). For example, given a Jenkins deployment in a project called `delivery`, and a build it needs to start in a project called `target`, the role bindings would be set up as follows:

```
[student@workstation ~]$ oc project
Using project "delivery" on server "https://master.lab.example.com:443".

[student@workstation ~]$ oc get sa jenkins
NAME      SECRETS   AGE
jenkins   2         3h

[student@workstation ~]$ oc -n target policy add-role-to-user edit \
> system:serviceaccount:delivery:jenkins
role "edit" added: "system:serviceaccount:delivery:jenkins"

[student@workstation ~]$ oc -n target get rolebinding/edit
NAME      ROLE      USERS     GROUPS    SERVICE ACCOUNTS   SUBJECTS
edit      /edit                         delivery/jenkins
```

The above configuration now allows Jenkins in project delivery to start a build (or deploy an application, etc.) in project `target`.

Guided Exercise 2.1: Run a Simple Pipeline using a Template

Exercise Overview

This lab will show you how to run a simple two-stage Jenkins pipeline.

Outcomes

After the exercise, there should be:

- A simple microservice running at the application URL.
- Evidence of having executed a Jenkins pipeline, rather than a simple build/deploy.
- Jenkins logs showing us how the pipeline executed.

Exercise Resources/Configuration

Lab Files Location:	`${HOME}/do388-simple-pipeline/`
Application URL:	`http://hello-simple-pipeline.apps.lab.example.com/api/hello`
Resources:	`hello-template.yml`

Before you begin...

Introductory notes as per usual (lab setup foo, etc.).

Steps

1. Create a new OpenShift project, call it `simple-pipeline`.

```
[student@workstation ~]$ oc new-project simple-pipeline
Now using project "simple-pipeline" on server "https://master.lab.example.com:443".

You can add applications to this project with the 'new-app' command. For example, try:

    oc new-app centos/ruby-25-centos7~https://github.com/sclorg/ruby-ex.git

to build a new example application in Ruby.
```

2. Create a new template for the pipeline.

   2.1. Create the template resource definition:

```
[student@workstation ~]$ oc create -f hello-template.yml
template.template.openshift.io/hello-pipeline created
```

   2.2. Examine the template:

```
[student@workstation ~]$ oc describe templates/hello-pipeline
Name:          hello-pipeline
Namespace:     simple-pipeline
Created: 57 seconds ago
```

```
...
(output omitted)
...
Objects:
    Service                               ${APP_NAME}
    ImageStream.image.openshift.io        ${APP_NAME}
    BuildConfig.build.openshift.io        ${APP_NAME}
    DeploymentConfig.apps.openshift.io    ${APP_NAME}
    Route.route.openshift.io              ${APP_NAME}
    BuildConfig.build.openshift.io        ${APP_NAME}-pipeline
```

2.3. Take a look at template parameters:

```
[student@workstation ~]$ oc process templates/hello-pipeline --parameters
NAME                          DESCRIPTION  GENERATOR   VALUE
APP_NAME                      The name...
SOURCE_REPOSITORY_URL         The URL...               http://services.lab.example.com/...
SOURCE_REPOSITORY_CONTEXT     The name...              standalone
MAVEN_MIRROR_URL              The URL...               http://services.lab.example.com:...
JDK_IMAGE_VERSION             The vers...              1.5
APP_HOSTNAME                  The host...
GITHUB_WEBHOOK_SECRET         The GitH...  expression  [a-z]{1}[a-z0-9]{31}
GENERIC_WEBHOOK_SECRET        The Gene...  expression  [a-z]{1}[a-z0-9]{31}
```

Note that the only parameters without a value are `APP_NAME` and `APP_HOSTNAME`, and even the latter will be generated if empty, so we only need to specify the application name.

3. Create a new application using the new template.

3.1. Create a new application with the name of `hello`:

```
[student@workstation ~]$ oc new-app --template=hello-pipeline -p APP_NAME=hello
--> Deploying template "simple-pipeline/hello-pipeline" to project simple-pipeline

     Java App + Pipeline
     ---------
     Creates a simple application build/deploy pipeline, based on the OpenJDK
     image, and all the associated resources.
...
(output omitted)
...
--> Creating resources ...
    service "hello" created
    imagestream.image.openshift.io "hello" created
    buildconfig.build.openshift.io "hello" created
    deploymentconfig.apps.openshift.io "hello" created
    route.route.openshift.io "hello" created
    buildconfig.build.openshift.io "hello-pipeline" created
--> Success
```

```
    Use 'oc start-build hello' to start a build.
    Access your application via route 'hello-simple-pipeline.apps.lab.example.com'
    Build scheduled, use 'oc logs -f bc/hello-pipeline' to track its progress.
    Run 'oc status' to view your app.
```

Notice how a build is automatically scheduled. Also, notice this will be a pipeline build:

```
[student@workstation ~]$ oc get builds
NAME               TYPE               FROM      STATUS    STARTED   DURATION
hello-pipeline-1   JenkinsPipeline              New
```

3.2. Notice how a jenkins pod is automatically deployed, even though nothing created it specifically:

```
[student@workstation ~]$ oc get pods
NAME               READY     STATUS              RESTARTS   AGE
jenkins-1-deploy   1/1       Running             0          3s
jenkins-1-l2vcl    0/1       ContainerCreating   0          0s
```

The above is due to a trigger that creates a Jenkins deployment every time a `BuildConfig` with a strategy of `JenkinsPipeline` is created:

```
[student@workstation ~]$ oc get bc
NAME             TYPE              FROM      LATEST
hello            Source            Git       0
hello-pipeline   JenkinsPipeline             1
```

3.3. Notice how both the `BuildConfig` and `DeploymentConfig` for application `hello` have all their triggers disabled (however, they are present):

```
[student@workstation ~]$ oc set triggers bc/hello
NAME                TYPE     VALUE                                    AUTO
buildconfigs/hello  config                                            false
buildconfigs/hello  image    openshift/redhat-openjdk18-openshift:1.5 false

[student@workstation ~]$ oc set triggers dc/hello
NAME                     TYPE     VALUE                  AUTO
deploymentconfigs/hello  config                          false
deploymentconfigs/hello  image    hello:latest (hello)   false
```

The most important of the above triggers is the `ImageChange` trigger in the `DeploymentConfig` - without it, any change to the `ImageStreamTag` of `hello:latest` would fail to update the application container's source image. It must be present, but disabled.

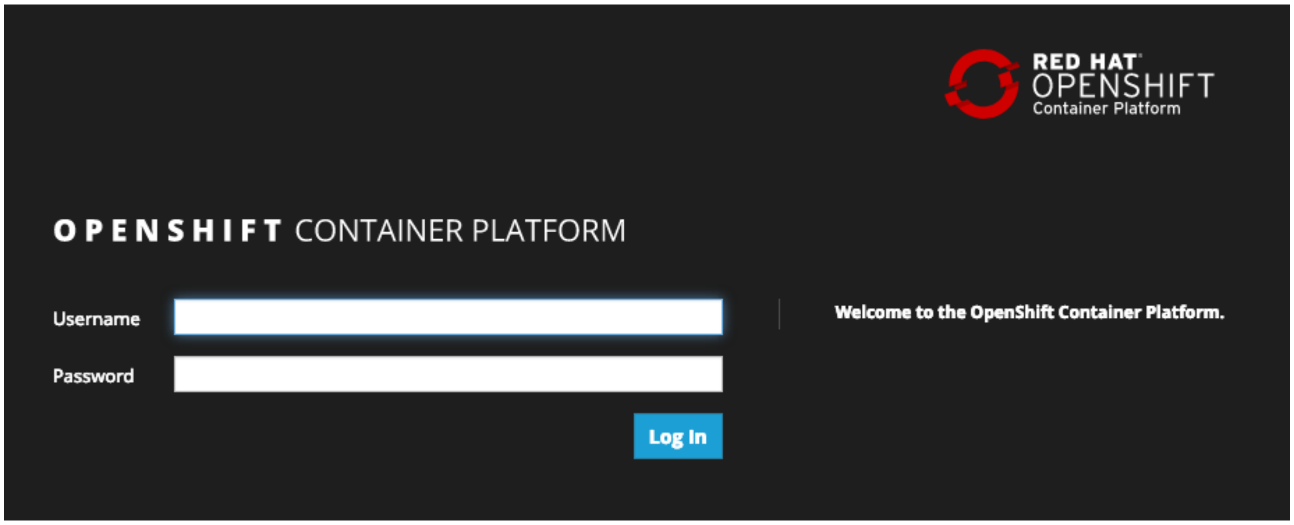
3.4. Notice how all the active triggers are on the `BuildConfig` for the pipeline:

```
[student@workstation ~]$ oc set triggers bc/hello-pipeline
NAME                         TYPE     VALUE                                      AUTO
buildconfigs/hello-pipeline  config                                              true
buildconfigs/hello-pipeline  image    openshift/redhat-openjdk18-openshift:1.5   true
```

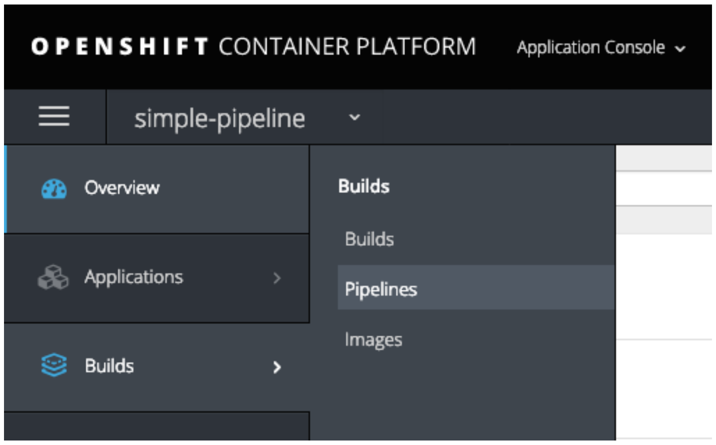
```
buildconfigs/hello-pipeline  webhook  <secret>
buildconfigs/hello-pipeline  github   <secret>
```

4. By now, a pipeline build should already be running. Inspect it, and its logs, from the OpenShift Application Console.

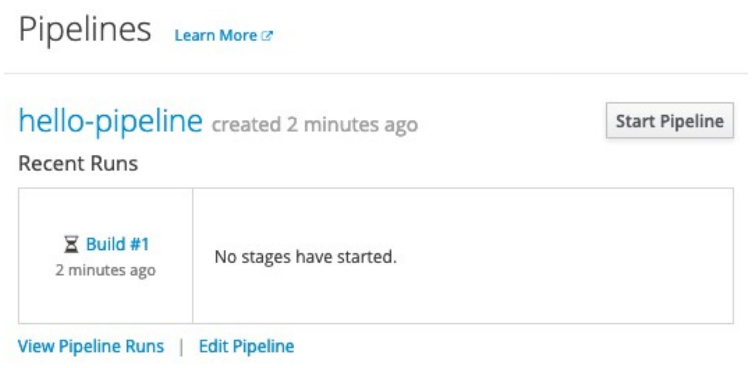
4.1. Log in to the OpenShift Application Console:


RED HAT OPENSHIFT Container Platform
OPENSHIFT CONTAINER PLATFORM
Username
Password
Log In
Welcome to the OpenShift Container Platform.

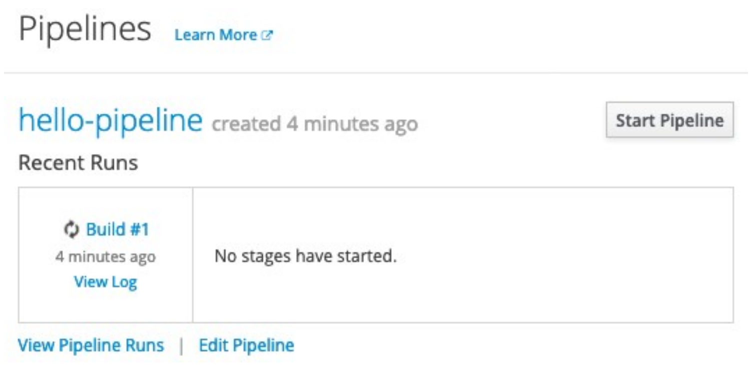
4.2. Navigate to Builds/Pipelines and inspect the pipeline build having started:



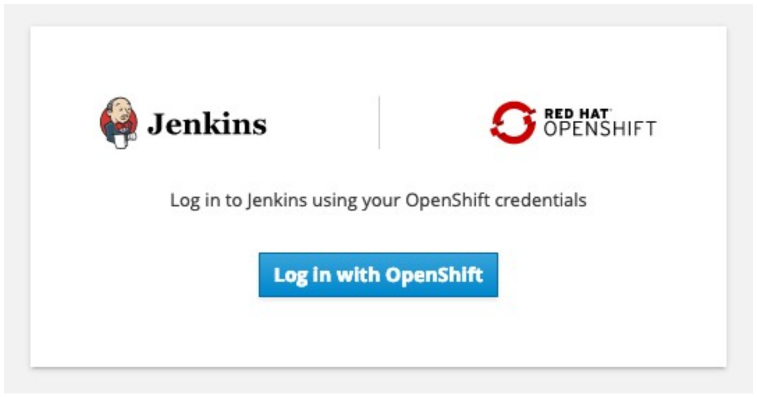
It may take a couple of minutes for the pipeline to run, depending on how long it takes for Jenkins to fully start up:



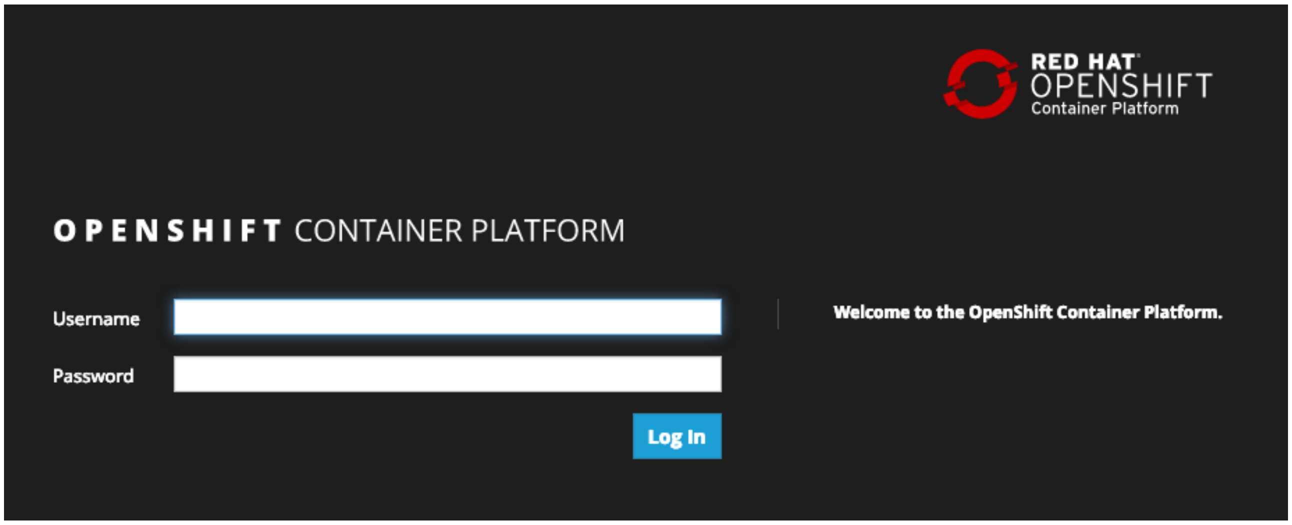
4.3. Once you see the View Log link appear, click it:



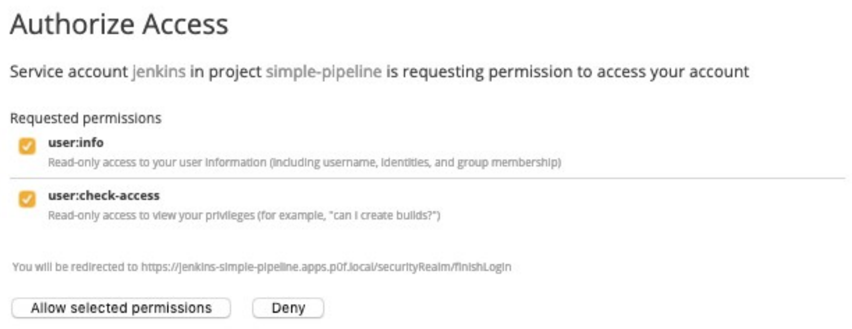
4.4. A new browser tab will open, and you will be redirected to the Jenkins web console. Confirm any security exceptions that might occur due to a self-signed certificate. When asked to log in using OpenShift authentication, click “Log in with OpenShift” to proceed:



4.5. Enter your username and password again when prompted:


RED HAT OPENSHIFT Container Platform
OPENSHIFT CONTAINER PLATFORM
Username
Password
Log In
Welcome to the OpenShift Container Platform.

4.6. You will be redirected back to Jenkins, but asked to allow it access to your information:



Authorize Access

Service account jenkins in project simple-pipeline is requesting permission to access your account

Requested permissions

- user:info
  Read-only access to your user information (including username, identities, and group membership)
- user:check-access
  Read-only access to view your privileges (for example, "can I create builds?")

You will be redirected to https://jenkins-simple-pipeline.apps.p0f.local/securityRealm/finishLogin

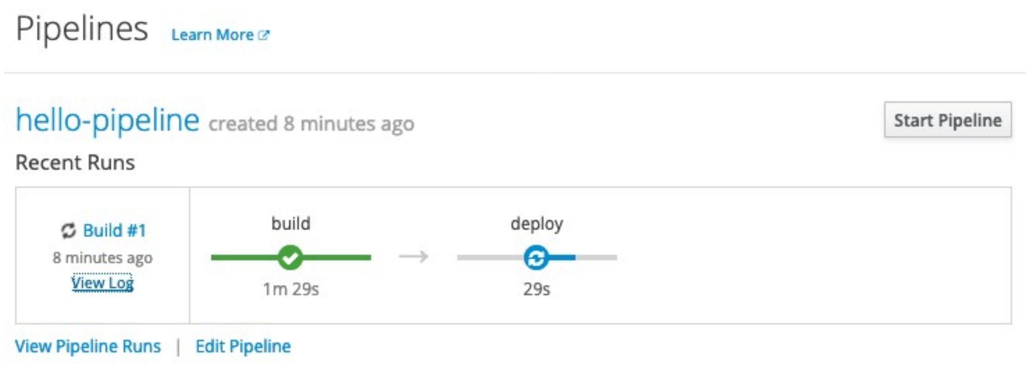
Confirm your agreement by clicking on "Allow selected permissions".

4.7. Finally, you will be presented with an ongoing record of operation progress in the pipeline, *the pipeline log*:

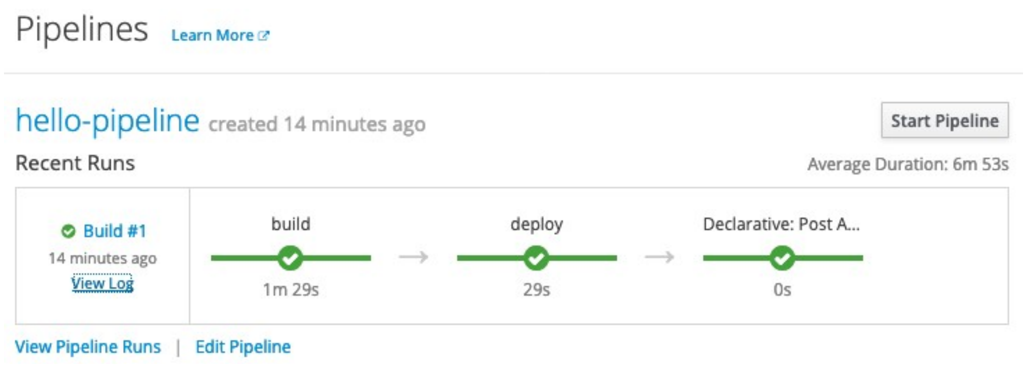
```
OpenShift Build simple-pipeline/hello-pipeline-1
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/jobs/simple-pipeline/jobs/simple-pipeline-
hello-pipeline/workspace
[Pipeline] {
[Pipeline] stage
...
(output omitted)
...
Pipeline] _OcAction
[logs:build/hello-1] Cloning "http://services.lab.example.com/hello-microservices/"
[logs:build/hello-1]       Commit:     1a447ca634356670f5cd5f55b199697f1080c18a
[logs:build/hello-1]       Author:     Grega Bremec <gbremec@redhat.com>
[logs:build/hello-1]       Date: Mon Feb 4 15:24:20 2019 +0200
...
```

```
(output omitted)
...
[logs:build/hello-1] Pushing image
docker-registry.default.svc:5000/simple-pipeline/hello:latest ...
[logs:build/hello-1] Pushed 5/6 layers, 84% complete
[logs:build/hello-1] Pushed 6/6 layers, 100% complete
[logs:build/hello-1] Push successful
[logs:build/hello-1]
[Pipeline] readFile
[Pipeline] readFile
[Pipeline] _OcAction
[Pipeline] }
[Pipeline] // script
Post stage
[Pipeline] echo
BUILD result: SUCCESS
...
(output omitted)
...
Pipeline] _OcAction
[rollout:latest:deploymentconfig/hello] deploymentconfig.apps.openshift.io/hello
rolled out
[rollout:latest:deploymentconfig/hello]
[Pipeline] readFile
[Pipeline] readFile
[Pipeline] _OcAction
[rollout:status:deploymentconfig/hello] Waiting for rollout to finish: 0 out of 1 new
replicas have been updated...
...
(output omitted)
...
```

4.8. Close the Jenkins browser tab and observe the pipeline finish in the OpenShift Application Console:



Once the pipeline build finishes, it will also display the average execution time per pipeline run:



Log out of the OpenShift Application console.

5. Back at the command prompt, test the application.

```
[student@workstation ~]$ oc get routes/hello
NAME    HOST/PORT                                   PATH   SERVICES   PORT   TERMINATION
WILDCARD
hello   hello-simple-pipeline.apps.lab.example.com         hello      8080
None
[student@workstation ~]$ curl hello-simple-pipeline.apps.lab.example.com/api/hello
hello
```

6. Clean up.

```
[student@workstation ~]$ oc delete project simple-pipeline
project.project.openshift.io "simple-pipeline" deleted
```

This concludes this practice exercise - you have successfully executed a simple pipeline.

Section 2.3: Writing Pipelines

There are two types of pipeline syntax in Jenkins, the so-called “scripted”, which is older, and is being replaced by the newer “declarative” syntax. Both forms of syntax use a simplified version of the Groovy language to express the instructions to Jenkins.

The biggest difference between the two is that the scripted syntax is more low-level, general-purpose, and therefore requires a more intimate knowledge of Jenkins internals, whereas the declarative syntax is easier (and safer) to use, if sometimes at the expense of slightly lesser control. Since our OpenShift Jenkins pipelines will manipulate OpenShift, and not Jenkins, resources, the declarative syntax is more suitable for our purposes.

In either case, the pipeline definition must be placed into the `jenkinsfile` attribute of the `BuildConfig` that defines it.

Defining the JenkinsPipeline BuildConfig

Let’s have another look at the `BuildConfig` object that carries the pipeline payload.

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: simple-pipeline
    name: simple-pipeline
  name: simple-pipeline
spec:
  source:                    (1)
    type: None
  output: {}                 (1)
  runPolicy: Serial          (2)
  strategy:
    type: JenkinsPipeline    (3)
    jenkinsPipelineStrategy: (3)
      jenkinsfile: |-        (4)
        node('maven') {
          stage('build') {
            // perform application build here
          }
          stage('test') {
            // this is where we do unit tests
            // also, tag the latest image as :tested
          }
          stage('deploy') {
            // deploy the tested image to next stage
          }
        }
  triggers:                  (5)
  - generic:
      secret: 0svtAKZuXj4GnB7uQ0bh42ZIFark9h
    type: Generic
```

1. A `JenkinsPipeline` build has no input and no output. It orchestrates other resources that do.
2. The `runPolicy` attribute in pipelines should always be set to `Serial` or `SerialLatestOnly` to prevent concurrency conflicts.
3. The type attribute of `BuildConfig`'s strategy is set to `JenkinsPipeline` and has only one required co-attribute - `jenkinsPipelineStrategy`.
4. The `jenkinsPipelineStrategy` attribute, in turn, also only requires one sub-attribute - `jenkinsfile`, which is to contain the definition of the pipeline that Jenkins will automatically import.
5. Just like any other `BuildConfig`, `JenkinsPipeline` BCs can also have triggers. They can be webhook triggers of any type, so instead of configuring your Git server to start a S2I build upon receiving a patch, you can easily configure it to start a pipeline. Additionally, a pipeline can monitor `ImageStream`s as well, so a change in the upstream base image can notify its `ImageChangeTrigger` and automatically start it.

Most other usual `BuildConfig` attributes do not apply to a `JenkinsPipeline`, as a Jenkins pod running in the same project will automatically import any changes to the `BuildConfig` and adjust the pipeline definition in Jenkins accordingly. Attributes like resource `requests` and `limits`, `nodeSelector`, and even `completionDeadlineSeconds`, therefore do not make any sense as there will be no additional builder pods for the pipeline - Jenkins is in charge of everything.

Writing Jenkinsfile Pipelines

A typical Jenkins pipeline will have several stages, each of which define some execution steps that need to be taken. In a very simple example, a pipeline definition would look like this:

```
pipeline {                                      (1)
  agent any                                     (2)
  stages {
    stage(‘log start’) {                        (3)
      steps {                                   (4)
        echo ‘starting pipeline execution’      (5)
      }
    }
    stage(‘ask the user’) {
      steps {
        input(message: ‘Should we continue?’, ok: ‘Absolutely!’)
      }
    }
  }
  post {                                        (6)
    success {
      echo ‘completed successfully’
    }
    aborted {
      echo ‘user interrupted execution’
    }
    always {
      echo ‘goodbye’
      }
    }
  }
```

1. The entire pipeline is placed into a closure. There can only be one pipeline closure per `jenkinsfile`.
2. If the pipeline is to be executed by master Jenkins executor, this value should be `any`, otherwise it should contain a definition of the agent node. We will discuss pipeline syntax in more detail later.
3. Each of the stages must have a unique name. It is a terminal error for two stages to have the same name.
4. Each stage contains a series of steps that need to be taken.
5. There are literally hundreds of pipeline steps that can be configured. Most are implemented by plugins.
6. The pipeline may conclude with a post-execution section, where it is possible to act upon the overall status not just of the current pipeline run, but also the outcome based on previous executions. Post-blocks such as `changed`, `fixed`, `regression`, `unstable`, and others can be used to reason over previous results of pipeline runs.

The above example uses the cleaner, *declarative* Jenkins pipeline syntax. It is possible to define a conceptually similar pipeline using a much more condensed scripted syntax, but the condensed form takes away a significant amount of readability and uses low-level control structures instead of the cleaner declarative behavior:

```
  node {                                    (1)
    stage('log start') {                    (2)
      echo 'starting pipeline execution'
    }
    stage('ask the user') {
      try {                                 (3)
        input(message: 'Should we continue?', ok: 'Absolutely!')
        echo 'completed successfully'
      } catch (exc) {
        echo 'user interrupted execution'
        throw exc
      } finally {
        echo 'goodbye'
      }
    }
  }
```

1. The scripted form starts with agent selection - either local to the Jenkins master, as in this case, or the node label selector should be supplied. Notice that in this case, `node`, and not `pipeline`, is the top-level closure.
2. Every subsequent section in the scripted pipeline is expected to be a stage containing a Groovy scriptlet.
3. There is no support for post-hooks. However, Groovy elements such as `try/catch/finally` can be used.

It is possible (and very common) to combine the declarative and scripted syntax by using a `script` closure in a declarative pipeline, for cases when a plugin was not yet adapted to declarative mode, or when simple low-level operations must be performed. We will discuss this in a later section.

Because of the differences between standard and OpenShift Jenkins pipelines, there are many pipeline concepts that are irrelevant in OpenShift, such as SCM check-out conditions, trigger controls, build tool definitions, stages executed in parallel, etc. We will not be discussing those concepts here.

In OpenShift, we essentially want to invoke a series of OpenShift operations on a number of resources, potentially across several different projects in a cluster (or even several clusters), and sometimes allow the user to supply some dynamic configuration data (such as commit ID of the build). We might require some user input in the midst of execution (such as an approval to promote the application to another stage). After each stage, and at the end of pipeline, we may want to reason over the outcomes.

Basic Declarative Pipeline Syntax

As mentioned above, each pipeline is a closure which consists of up to five sections:

- agent selection,
- environment variables,
- execution options,
- definition of stages, and
- post-lifecycle handlers.

Of the above, only the agent selection and stage definitions are mandatory.

Agent Selection

Agents that execute the pipeline are defined using the `agent` keyword, which can be followed by a number of different selectors. Each stage can use its own agent, but normally, the entire pipeline is entrusted to a single agent.

The common selectors in OpenShift are:

Agent Selector	Effect on Execution
`agent none`	Do not select an agent. Each stage must provide its own agent definition. In OpenShift, this means each stage will execute in a separate pod. With this global option, it is an error for a stage not to provide an agent definition.
`agent any`	Choose any available agent. In OpenShift, this means the embedded executor pool inside the Jenkins pod will execute the relevant stages.

`agent { label ‘maven’ }`
`agent { node { label ‘maven’ } }`
	Both forms select an agent which is labeled in Jenkins configuration as `maven`. While equivalent in effect, the second form is considered cleaner and allows for a few additional per-node options, which are not particularly useful in OpenShift context (such as `customWorkspace`).

There are other available selectors such as `docker` and `dockerfile`, but they are very seldom (if ever) useful in OpenShift.

Environment Variables

The `environment` closure can be used to define environment variables that are then referenced by steps that need to access them. Scriptlets are free to reset those values. Just like with agents, environment variables can be defined at the global pipeline scope, or for each individual stage. In the latter case, their visibility is only in the scope of the stage they are defined in.

```
pipeline {
  agent { label 'maven' }
  environment {
    PIPE_NAME = "sample pipeline"
  }
  stages {
    stage('start a build') {
      environment {
        STAGE = "build"
      }
      steps {
        // Both PIPE_NAME and STAGE are available here
        echo “Pipeline ${PIPE_NAME} running stage ${STAGE}...”
        ...
      }
    }
    stage(‘run the tests’) {
      steps {
        // PIPE_NAME is available here, but STAGE is null
        echo “Pipeline ${PIPE_NAME} running stage ${STAGE}...”
        ...
      }
    }
  }
}
```

The above pipeline demonstrates environment variable scoping in a simple, two-stage scenario. Because the `PIPE_NAME` variable is declared at the pipeline level, it is available in all stages, however, the `STAGE` variable is only available in the first stage due to its stage-scoped declaration.

Execution Options

Most of the execution options are meant to control pipeline execution in standalone Jenkins deployments outside of OpenShift, but a couple of them are useful in OpenShift context as well. They can be specified both at the pipeline and individual stage level using the `options` closure.

Option	Meaning

`quietPeriod(t:Integer)`
`quietPeriod(time: t:Integer, unit: u:String)`
	Only valid at pipeline scope. Any number of consecutive build requests within the specified period will only trigger one build. The two forms accept either seconds, or a period and unit using named parameters. Units can be either of: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, and `DAYS`.

Option	Meaning

`timeout(t:Integer)`
`timeout(time: t:Integer,`
`        unit: u:String)`
	If used at the pipeline scope, aborts the entire pipeline unless finished within the specified amount of time. If used at the stage scope, it sets a timeout for a single stage. The two forms accept either seconds, or a period and unit using named parameters. See `quietPeriod` for units. Also, see the `timeout` closure below.

`retry(n:Integer)`	If the entire pipeline or a single stage fail, this will attempt to re-run them a specified number of times before giving up.

Stage Definition

The `stages` closure in the pipeline definition must consist of one or more named stages which may, as mentioned above, contain some execution options and define some environment variables, but can also contain conditional execution clauses and post-execution steps (we will look at both of these in later sections).

Most importantly, the stages define a series of `steps` which must to be taken in order to complete each stage. More often than not, in addition to performing actions, these steps would contain some logic to ascertain whether the actions completed successfully, and some error handling logic to prevent non-fatal failures from interrupting the stage (and consequently, pipeline).

A stage definition outline looks like this:

```
stages {
  stage('must have a name') {
    environment { ... }
    options { ... }
    when { ... }
    steps {
      echo “Starting the stage...”
      // do things
    }
    post { ... }
  }
}
```

In a declarative pipeline, steps can include some basic directives, such as `echo`, `mail`, `sleep`, or `error`, but also invoke operations enabled by various plugins that support declarative mode (such as the old OpenShift Jenkins plugin). Some useful basic directives and closures include:

Directive / Closure	Purpose

`echo(...)`
`println(...)`
`printf(...)`
	Evaluate the expression according to Groovy syntax rules and print it on standard output. Within Jenkins, this means it becomes a part of the pipeline build log.

`mail(subject: s:String,`
`     body: b:String[,`
`     from: f:String,`
	Sends an email with the subject of `s` and body of `b`. All other parameters are optional. Recipient parameters (`to`, `cc`, and `bcc`) are comma-separated lists of

Directive / Closure	Purpose

`    to: t:String,`
`    replyTo: r:String,`
`    cc: c:String,`
`    bcc: bc:String,`
`    mimeType: mt:String,`
`    charset: cs:String])`
	recipients, but it is a runtime error if none of them contain any addresses.
	The defaults for `from` and `replyTo` are the globally configured defaults for the Jenkins instance. The defaults for `mimeType` and `charset` are `text/plain` and `UTF-8`, respectively. The defaults for `to`, `cc`, and `bcc` are empty lists.
	Given that the mail operation is attempted within a container, and Jenkins expects a SMTP server to be running at `localhost:25`, this step will require some additional Jenkins configuration before it works correctly.

`sleep(time: t:Integer,`
`      unit: u:String)`
	Pauses execution of the stage and does nothing for `t` of time units `u`. Unit is any of `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, and `DAYS`.

`timeout(time: t:Integer,`
`        unit: u:String) { ... }`
	Anything within the enclosing block is given `t` of time units `u` to complete. See `sleep` above for units. It is an error if the block does not complete within a specified amount of time.

`retry(n:Integer) { ... }`	Anything within the enclosing block is attempted up to `n` times if unsuccessful.
`error([message: m:String])`	Throws an exception, optionally with a message of `m`. Functionally equivalent to `throw new Exception()`, but does not print a stack trace in the logs.
`catchError { ... }`	If an error occurs within the enclosing block, sets the build result to `FAILURE` or `ABORTED` (depending on the error caught), but continues pipeline execution. Usually, a better idiom for handling errors is a `try/catch/finally` statement.
`waitUntil { ... }`	The body of this closure is executed repeatedly until it returns `true`. The number of repetitions is infinite, but with a back-off between retries. Any exception will immediately terminate the closure and propagate upwards.
`script { ... }`	Declares a scripted section of a stage. Within such blocks, it is allowed to use most Groovy syntax such as conditionals, `try/catch/finally` blocks, variable declarations and assignments, loops, etc., which are otherwise forbidden in declarative syntax. Additionally, global variables such as the OpenShift Client Jenkins plugin are only available within a `script` closure.

`try { ... }`
`catch (e) { ... }`
`finally { ... }`
	As opposed to the `catchError` closure, any failure that occurs within the `try` block will cause the execution to jump to the `catch` block, and it is up to the code therein to decide whether to re-throw the exception or suppress it. The `finally` block is executed last, regardless of whether there was an error or not. Only allowed within a `script` closure.

`if (...) { ... } else { ... }`	A classic conditional statement. Only allowed within a `script` closure.

Directive / Closure	Purpose
`def x = foo()`	A classic assignment with type inference. Only allowed within a `script` closure.

One of the most important bits of information when doing error recovery using `try/catch/finally` is to be aware of different types of useful exceptions that can be thrown in your pipelines, as that allows us to be selective about how to handle the error (for example, the `org.jenkinsci.plugins.workflow.steps.FlowInterruptedException` signifies that most probably a `timeout` clause interrupted a waiting task, or a user aborted a prompt).

Execution Status

Each stage will have been assigned a status after its completion. The status generally depends on the outcome of the last executed step in the stage, but can also be modified programmatically. The overall status of pipeline execution generally depends on the outcome of the last executed stage, unless the result has been modified programmatically at some point.

The implicit variable `currentBuild` is available in `script` blocks, and one can use it to reason about the overall state of the pipeline execution, the state of the current stage, and the duration of current execution run. It is an object of type `org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper`, and has the following useful properties:

Property	Content
`currentBuild.result`	The overall result of the current pipeline run. See below for possible values.
`currentBuild.currentResult`	The result of the current stage execution. See below for possible values.
`currentBuild.duration`	The amount of time spent running the pipeline so far, in milliseconds.
`currentBuild.durationString`	Human readable version of the above (i.e. “x minutes y seconds and counting”).

It is possible to programmatically reset the outcome of a stage by manually setting the value of `currentBuild.result` to one of five possible outcomes. Depending on the value used, the pipeline will be marked as successful, failed, or aborted. Consult the following table for details:

Outcome	Success?	OCP Build Status	Usual Cause(s)
`SUCCESS`	yes	Complete	The stage/pipeline completed successfully.
`ABORTED`	no	Cancelled	Manual user interruption or a timeout. Terminates the stage immediately and skips all remaining stages.
`FAILURE`	no	Failed	There was an error during execution of one of the steps. Terminates the stage immediately and skips all remaining stages.
`UNSTABLE`	no	Failed	Some non-functional tests failed, the application is not to be trusted.
`NOT_BUILT`	yes	Pending	Build skipped due to policy (e.g. no builds on Friday). Remains in Pending state until cancelled manually.

Note that contrary to the usual behavior of `ABORTED` and `FAILURE` outcomes, setting them manually does nothing to terminate the current stage and skip the rest of the pipeline. While the pipeline outcome is irreversibly set to one or the other, it keeps executing until it either finishes or an actual error is encountered.

Post-Execution Steps

Each stage, as well as the overall pipeline, can be configured to perform some additional steps based on their outcome. These steps can be configured in the `post` closure, which can contain a number of blocks, each of which will be executed if the execution outcome matches. If several blocks match, all of them will be executed. The blocks available in the `post` closure are:

Block	Matching pipeline status
`always { ... }`	Executes regardless of completion status, always first among the blocks.
`cleanup { ... }`	Executes regardless of completion status, always last among the blocks.

`success { ... }`
`failure { ... }`
`aborted { ... }`
`notBuilt { ... }`
`unstable { ... }`
	Only executes if the stage/pipeline finished with the corresponding status.

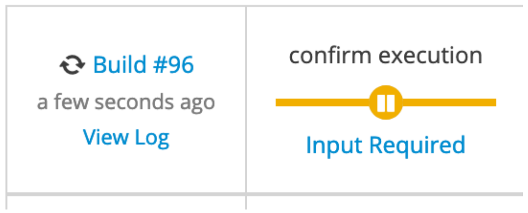
`changed { ... }`	Only executes if the outcome of the stage/pipeline is different than the last time.
`fixed { ... }`	Only executes if the outcome is `SUCCESS`, and the previous run resulted in either `FAILURE` or `UNSTABLE` (but not `ABORTED`).
`regression { ... }`	Only executes if the outcome is `FAILURE`, `UNSTABLE`, or `ABORTED`, and the previous run was a `SUCCESS`.

Making Pipelines Interactive

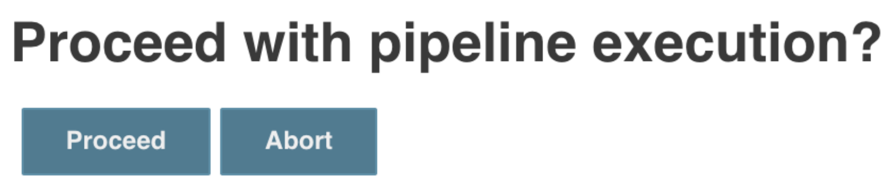
Any stage can use the `input` step that will pause the execution and wait for user interaction, by default simply presenting a message and giving an option to proceed or abort the stage (and thus, usually, pipeline). Consider the following example:

```
stage('confirm execution') {
  steps {
    input(message: "Proceed with pipeline execution?")
  }
}
```

The above step will appear as `Input Required` in the OpenShift web console, and wait for the user to click the link:



Once clicked, the user is required to login to Jenkins console, and the corresponding input form is presented:


Proceed with pipeline execution?

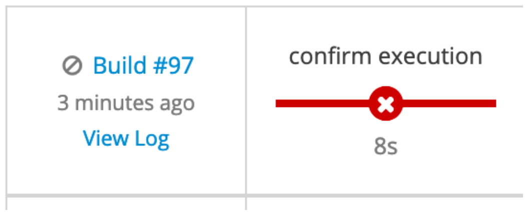
If the user confirms the form by clicking `Proceed`, the pipeline log will report success:

```
[Pipeline] {
[Pipeline] stage
[Pipeline] { (confirm execution)
[Pipeline] input
Proceed with pipeline execution?
Proceed or Abort
Approved by johndoe
[Pipeline] }
```

In the other case, execution status of the entire pipeline is set to `ABORTED` and the remaining stages are skipped:

```
[Pipeline] {
[Pipeline] stage
[Pipeline] { (confirm execution)
[Pipeline] input
Proceed with pipeline execution?
Proceed or Abort
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (subsequent stage)
Stage "subsequent stage" skipped due to earlier failure(s)
[Pipeline] }
...
(output omitted)
...
[Pipeline] // node
[Pipeline] End of Pipeline
Rejected by johndoe
Finished: ABORTED
```

The OpenShift web console will also show the stage as failed and the pipeline status as aborted:



It is possible to customize the input dialog somewhat using additional parameters, such as:

Option	**Description**
`message:String`	The message to be displayed in the input dialog.
`ok:String`	The text to display on the confirmation button. Defaults to `Proceed`.
`parameters:Map`	Additional parameters to present to the user in the form. See below.

Optionally, the `input` step can ask for additional information by means of parameters, which are returned as an associative array (Groovy map) upon completion of the step. If there is only one parameter, its value is returned directly. Remember that if assignments are to be used, the entire `input` step must be enclosed in a `script` block.

There are several different types of parameters that can be used in the `input` step. Almost all of them accept the same options, of which only `name` is mandatory:

- `name`, specifying the name of the map key to store the result into
- `description`, displayed in the form underneath the parameter prompt, as an additional instruction to the user
- `defaultValue`, specifying the default value to use in case no input is provided (not valid for `file` and `choice`)

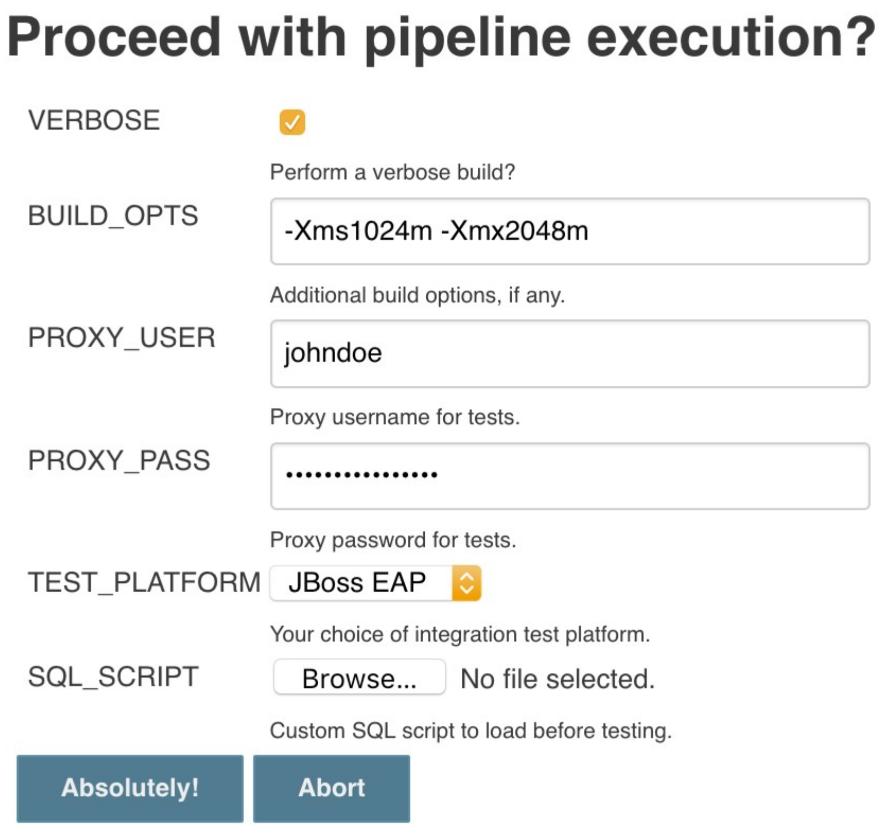
For `choice`, instead of using `defaultValue`, one must provide an array of `choices` that will be rendered in a drop-down menu, whereby the first element in the array represents the default value. The parameters are rendered as HTML elements according to their type specification as follows:

Parameter Type	**Description**
`booleanParam()`	Renders as a checkbox in the displayed form.
`string()`	The user is presented with an input field for a single line of text.
`text()`	The form renders a text area, where multiple lines of text can be entered.
`password()`	Same as `string()`, but with masked input.
`choice()`	A drop-down menu of choices, one of which may be selected.
`file()`	A file upload button. The file is stored in Jenkins agent's workspace and the map key specified as the `name` contains the absolute location of that file after upload.

Consider the following, slightly more advanced, input step:

```
stage('confirm execution') {
  steps { script {
    def response = input(
            message: "Proceed with pipeline execution?",
            ok: "Absolutely!",
            parameters: [
              booleanParam(name: "VERBOSE", defaultValue: true,
                          description: "Perform a verbose build?"),
              string(name: "BUILD_OPTS",
                     description: "Additional build options, if any."),
              string(name: "PROXY_USER",
                     description: "Proxy username for tests."),
              password(name: "PROXY_PASS",
                       description: "Proxy password for tests."),
              choice(name: "TEST_PLATFORM",
                     choices: [ "Thorntail", "JBoss EAP", "SpringBoot" ],
                     description: "Your choice of integration test platform."),
              file(name: "SQL_SCRIPT",
                   description: "Custom SQL script to load before testing."),
            ])
      echo "GOT INPUT: ${response}"
      echo "Test platform will be ${response.TEST_PLATFORM}."
} } }
```

The above pipeline will render as a form in the Jenkins web console:



Proceed with pipeline execution?

VERBOSE	✓ Perform a verbose build?
BUILD_OPTS	-Xms1024m -Xmx2048m Additional build options, if any.
PROXY_USER	johndoe Proxy username for tests.
PROXY_PASS	•••••••••••••••• Proxy password for tests.
TEST_PLATFORM	JBoss EAP Your choice of integration test platform.
SQL_SCRIPT	Browse... No file selected. Custom SQL script to load before testing.

Looking at the Jenkins pipeline logs, it is possible to see the results of the above form thanks to the `echo` step:

```
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (confirm with file upload prompt)
[Pipeline] script
[Pipeline] {
[Pipeline] input
Input requested
Approved by johndoe
[Pipeline] echo
GOT INPUT: [BUILD_OPTS:-Xms1024m -Xmx2048m,
SQL_SCRIPT:/var/lib/jenkins/jobs/myproject/jobs/myproject-pipeline/builds/51/SQL_SCRIPT,
TEST_PLATFORM:JBoss EAP, VERBOSE:true, PROXY_PASS:secretpassword, PROXY_USER:johndoe]
[Pipeline] echo
Test platform will be JBoss EAP.
[Pipeline] }
[Pipeline] // script
[Pipeline] }
```

Bear in mind that the results of the interaction, when stored into a variable, are scoped to this stage only. Should there be a use for them in a different stage, they must be stored into a pipeline-scoped environment variable:

```
pipeline {
    agent any
    environment {                                         (1)
      VERBOSE_BUILD = true
      BUILD_TIMEOUT_MIN = 30
    }
    stages {
      stage(‘user input’) { steps { script {              (2)
        def resp = input(message: “Please provide input:”,    (3)
                         parameters: [
                           booleanParam(name: “VERBOSE”, description: “Verbose build?”),
                           choice(name: “TIMEOUT”, choices: [30, 20, 10],
                                  description: “Select build timeout”),
                         ])
        VERBOSE_BUILD = resp.VERBOSE                      (4)
        BUILD_TIMEOUT_MIN = resp.TIMEOUT                  (4)
      } } }
      stage(‘perform build’) {
        steps { script {                                  (5)
          timeout(time: BUILD_TIMEOUT_MIN, unit: “MINUTES”) { (6)
            if (VERBOSE) {                                (6)
              // perform a verbose build
            } else {
              // perform a quiet build
            }
          }
  } } } } }
```

1. Defining pipeline-scoped variables in the `environment` section allows us to propagate `input` to a higher scope.
2. Stage “`user input`” must use a `script` closure to perform assignment of `input` results to variable `resp`.
3. Request user to provide feedback and store the results into variable `resp`, which is a map (two results are passed).
4. Assign user feedback to pipeline-scoped variables.
5. Stage “`perform build`” must use a `script` closure to perform conditional execution using the `if` statement.
6. In a different stage, the `resp` variable is not visible (out of scope), so we refer to pipeline-scoped variables.

Another important thing to note is that `input` steps do not time out. Should there be a time limit for a user to provide feedback, the `input` step must be enclosed in a `timeout` block:

```
timeout(time: 60, unit: “SECONDS”) {
  input(message: “Please provide input”, ...)
}
```

If the user exceeds the time limit for a reply, the result of the stage is set to `ABORTED` and any subsequent stages are skipped. If the output is supposed to be optional, then a `try/catch/finally` block must be used to suppress the timeout exception:

```
timeout(time: 60, unit: “SECONDS”) {
  script {
    try {
      input(message: “Please provide optional input”, ...)
    } catch (exc) {
      echo “User input timed out: ${exc}”
    }
  }
}
```

Note that in the interest of safety, the above `catch` block should also check for the correct type of exception before suppressing it (timeouts, as mentioned before, cause exceptions of type `FlowInterruptedException`).

Conditional Stage Execution

The `when` closure in the definition of a stage can be used to express conditions under which the stage should be executed. If the conditions do not evaluate to `true`, the entire stage is skipped. The `when` clause accepts several different conditions, some of which only make sense in a standalone Jenkins deployment. The conditions that are useful in OpenShift include:

Condition	Description
`environment(name: n:VarNameString, value: v:StringExpr)`	Evaluates to `true` when the specified environment variable contains the specified value. The `value` parameter can be a Groovy expression string.
`equals(expected: e:LiteralValue, actual: a:FieldReference)`	Evaluates to `true` when the `expected` value is equal to the value contained in a variable or field reference referred to by `actual`.
`expression { ... }`	Evaluates to whatever the return value of a Groovy expression within the closure is. Note that the return value must be a `null` or boolean type to actually evaluate to `false`.
`not { ... }`	Inverts the return value of the nested condition.
`allOf { ... }`	Requires all of the nested conditions to evaluate to `true` in order for the overall result to be successful.
`anyOf { ... }`	Successful as long as at least one nested condition evaluates to `true`.

Conditional execution can be very useful for several different situations. For example, a non-terminal failure in a prior stage might disqualify the current one, but we do not want to completely abort the pipeline (i.e. run code quality and test coverage analysis even though unit tests failed, but do not deploy anything). Another example is rather straightforward use of input parameters as toggles for certain stages:

```
    pipeline {
      agent any
      environment {
        SKIP_INT = false                                   (1)
      }
      stages {
        stage(‘get user input’) {
```

```
        steps { script {
          def skip = input(message: “Please provide input:”,    (2)
                           parameters: [
                             booleanParam(name: “SKIP”, defaultValue: false,
                                          description: “Skip integration tests?”)
                           ])
          SKIP_INT = skip                                       (3)
        } }
      }
      // ...
      // other stages go here
      // ...
      stage(‘integration tests’) {
        when {
          not { expression { return SKIP_INT } }                (4)
        }
        steps {
          // perform integration tests
        }
      }
    }
  }
```

1. Defines a pipeline-scoped global variable to store the user feedback into (as we use it in a different stage).
2. Ask the user whether to skip the integration tests.
3. Promote the response to the pipeline-scoped variable (note that there is only one parameter for the `input` step, so its result is stored directly in variable `skip`, and not as a map entry of `skip.SKIP`, as would be the case with multiple parameters).
4. Evaluate the `SKIP_INT` variable to decide whether to run the stage or not.

There are obviously many other uses for conditional execution which will typically depend on the actual content of pipeline steps, and the logic validating their outcome. Remembering that propagating the outcome to a pipeline-scoped variable in order to be able to refer to it in a subsequent stage is the key.

Working with OpenShift

Between the 3.7 and 3.11 releases of OpenShift, there were two ways of invoking OpenShift operations from Jenkins pipelines - either using the old (and now deprecated) OpenShift Jenkins plugin, or the newer OpenShift Client Jenkins plugin.

A Bit of History

The older version of the plugin uses a Java JSON client to communicate with OpenShift, and despite the fact it is very easy to use, it is quite limited with regard to the number of features it can support and the agility with which it can follow OpenShift evolution. It is still very commonly used, however, so you will probably encounter it a lot. Here is a trivial example of three stages that build, test, and deploy an imaginary OpenShift application:

```
pipeline {
    agent { node { label 'maven' } }
    stages {
      stage('build') {
        steps {
          openshiftBuild(bldCfg: 'myapp-build', showBuildLogs: 'true')    (1)
        }
      }
      stage('test') {
        steps {
          openshiftBuild(bldCfg: 'myapp-tests', showBuildLogs: 'true')    (2)
          openshiftTag(srcStream: ‘myapp’, srcTag: ‘latest’,              (3)
                       dstStream: ‘myapp’, dstTag: ‘tested’)
        }
      }
      stage('deploy') {
        steps {
          openshiftDeploy(depCfg: 'myapp')                                (4)
        }
      }
    }
  }
```

1. This Jenkins step starts a new OpenShift build, defined in a `BuildConfig` called `myapp-build` in the current project. Presumably, the `BuildConfig` pushes the image into an `ImageStreamTag` `myapp:latest`. There is a large number of additional options for this step that allow us to specify an arbitrary OpenShift API endpoint and token, pick a custom commit ID to build, specify different build parameters, etc.
2. In the second stage, presumably the `myapp-tests` `BuildConfig` is configured such that it starts the unit tests instead of just a simple build and packaging of the application (for example, `mvn verify` instead of the Java-based S2I default of `mvn -DskipTests package`).
3. If the previous step in this stage was successful (i.e. if the unit tests succeeded), we will tag the current `myapp:latest` image with an additional tag, `myapp:tested`.
4. Finally, a new deployment is started, defined in the `DeploymentConfig` called `myapp` in the current project. Again, presumably this `DeploymentConfig` is deploying pod containers from an `ImageStreamTag` called `myapp:tested`, so it will only have deployed images that had passed the testing stage.

The OpenShift Jenkins plugin defines a number of other steps that allow us to create and delete arbitrary OpenShift resources, scale deployments, execute arbitrary commands in running pods, and verify builds, deployments, and services.

As a general rule, these steps will block the pipeline execution until the corresponding OpenShift operation completes, and the operation outcome propagates upwards and into the pipeline (i.e. if a build has failed, the pipeline step will fail at that point).

Invoking OpenShift Operations

The new OpenShift Client Jenkins plugin is a complete rewrite, and uses the `oc` binary built into Jenkins images. Maintaining feature parity with OpenShift releases is hence much simpler.

As a result of using the client binary, starting an OpenShift operation does not necessarily block the pipeline execution, and unless we tell the command to wait for the outcome, all operations are executed asynchronously. The stage steps would have to wait for the completion of an operation using some sort of control construct such as polling loops and timeouts.

The above old-style pipeline would have to be quite substantially rewritten using the new plug-in (note, some stages have been reformatted for brevity, but the pipeline is syntactically correct as it is):

```
pipeline {
  agent { node { label 'maven' } }
  stages {
    stage('build') {
      steps {
        script {                                                        (1)
          openshift.withCluster() {                                     (2)
            openshift.withProject() {                                   (3)
              def x = openshift.selector("bc/myapp-build").startBuild() (4)
              x.logs("-f")                                              (5)
    } } } } }
    stage('test') {
      steps { script {
        openshift.withCluster() { openshift.withProject() {
          def x = openshift.selector("bc/myapp-test").startBuild()
          x.logs("-f")
          if (x.object().status.phase == “Complete”) {                  (6)
            openshift.tag(“myapp:latest”, “myapp:tested”)
          }
    } } } } }
    stage('deploy') {
      steps { script {
        openshift.withCluster() { openshift.withProject() {
          openshift.selector("dc/myapp").rollout().latest()             (7)
          def x = openshift.selector("dc/myapp").related("pods")        (8)
          x.untilEach(1) {                                              (9)
            return (it.object().status.phase == "Running")
          }
    } } } } }
  }
}
```

1. The OpenShift Client Jenkins plugin is exposed as an implicit variable `openshift`, so it must be used in a `script` closure.
2. It is possible to use the plugin with multiple different OpenShift clusters - the cluster definition should be added to Jenkins configuration settings and then referred to by its name in the `withCluster` closure.
3. The `withProject` closure allows us to execute a series of operations in the same OpenShift project. An empty parameter list refers to the same project as the Jenkins pod is deployed to. Of course, normal `ServiceAccount` restrictions apply when trying to invoke operations in other projects.

4. Using a selector, obtain a reference to a `BuildConfig`, then invoke the `startBuild()` method on it. The result of the `startBuild()` method invocation is stored in variable `x`, and (rather expectedly) refers to a build.
5. Follow the build logs until the build finishes.
6. Only tag the `myapp` image as tested if the build running unit tests completed successfully.
7. Again, using a selector, obtain a reference to a `DeploymentConfig`, then invoke a rollout of its latest version.
8. Obtain references to all the pods related to the latest deployment of the application `myapp`.
9. In a loop requiring at least one desired pod, wait until all pods for this deployment are in state `Running`.

The non-blocking mode of the new plugin can be very convenient if several operations have to be performed in parallel, or even if certain operations span several pipeline stages. Unfortunately, this does amount to some boilerplate code even in cases where we do not require such advanced features (as in the above example).

As a consequence of asynchronous execution, the outcome of OpenShift operations does not always automatically propagate into the pipeline, so a negative outcome should be tested for and handled explicitly (for example by raising an `error`).

Manipulating OpenShift Resources

Using the implicit `openshift` variable, it is possible to manipulate existing OpenShift resources, or create new ones. The `openshift` object has the following useful methods for that:

Method	**Description**
`openshift.logLevel(l:Integer)`	Changes the log level for all subsequent operations (regardless of the cluster and/or project used) to l.
`openshift.verbose(b:Boolean)`	If true, changes the log level to 8. If false, sets it to 0.
`openshift.withCluster([c:String[, t:String]]) { ... }`	Without a parameter, uses the cluster that the Jenkins pod is deployed to, and the Jenkins service account credential. With a single parameter, uses one of the clusters manually defined in Jenkins configuration. If a second parameter is specified, it should refer to a credential (also defined in Jenkins configuration) that should be used to log in.
`openshift.withProject([p:String]) { ... }`	Without a parameter, selects the project the Jenkins pod is deployed to. With a parameter, uses project `p`. Any operation within the closure is executed in that project.
`openshift.newProject(n:String[, p...:String])`	Creates a new OpenShift project, but does not select it. Any additional parameters are passed verbatim to the `oc` tool.
`def s = openshift.newApp(p...:String)`	Invokes the `oc new-app` command. Any parameters are passed verbatim to the `oc` tool. Returns a static selector referring to any resources created by this command.

Method	Description
`def s = openshift.newBuild(p...:String)`	Invokes the `oc new-build` command. Any parameters are passed verbatim to the `oc` tool. Returns a static selector referring to any resources created by this command.
`def s = openshift.startBuild(p...:String)`	Starts a build. The parameters must at the very minimum be selective enough to specify a build to start (such as `bc/foo` or `--from-build=foo-1`). Returns a static selector referring to the build (and any other objects) created by the command.

`def s = openshift.create(verb:String[, p...])`
`def s = openshift.create(m:Map[, p...])`
`def s = openshift.create(l:List<Map>[, p...])`
`def s = openshift.create(json:String[, p...])`
`def s = openshift.replace(...)`
`def s = openshift.apply(...)`
	The equivalents of `oc create`, `oc replace`, and `oc apply` commands. They all have three different signatures, each of which accepts a different representation of resources that need to be created or updated.
	Additionally, `create()` supports a form where a sub-verb with its corresponding parameters can be supplied directly (such as `create(“serviceaccount”, “jenkins”)`).
	They all return a static selector referring to any resources that were just created or updated. See below.

`def m = openshift.process(t:String[, p...])`
`def m = openshift.process(json:String[, p...])`
`def m = openshift.process(m:Map[, p...])`
	The equivalent of `oc process` command. In the first form, the template is referred to by name (it must already exist in OpenShift), in the second form it is passed as a string literal, and in the third, it is a Groovy map containing unmarshalled JSON representation of the template. Returns a Groovy map representing the resulting resources.

`openshift.exec(p...:String)`
`openshift.idle(p...:String)`
`openshift.policy(p...:String)`
`openshift.run(p...:String)`
`openshift.secrets(p...:String)`
`openshift.set(p...:String)`
`openshift.tag(p...:String)`
	These are all literal equivalents of corresponding `oc` commands. Any parameters are passed to the `oc` tool verbatim.

As mentioned above, the `create()`, `replace()`, `apply()`, and `process()` methods have several different signatures where OpenShift resources can be represented using three different means.

The simplest one is where a JSON or YAML string is passed verbatim as the first parameter, and any additional parameters to the command follow it. Consider this simplified example:

```
def yaml = “””apiVersion: v1
kind: Pod
metadata:
  name: apod
spec:
  containers:
  - name: mycontainer
    image: myproject/myimage:latest
"""
def s = openshift.create(yaml, "--loglevel=5")
```

An interesting variation of the above is possible when the JSON or YAML definition is stored in a file:

```
def s = openshift.create(readFile(file:"definitions.json"), "--loglevel=5")
```

The above string representation of a pod can be converted to a Groovy map and used with the second form:

```
def map = [ apiVersion: "v1", kind: "Pod",
            metadata: [ name: "apod" ],
            spec: [
              containers: [
                [ name: "mycontainer", image: "myproject/myimage:latest",
                  ports: [ [ name: "http", protocol: "TCP", containerPort: 8080 ] ] ] ]
          ] ] ]
def s = openshift.create(map, "--loglevel=5")
```

The above two forms can also be used with the `process()` method, but the resource described in the first parameter must be a template in that case, as `oc process` requires it.

Finally, if multiple resources are to be created or updated, the first parameter can be a list of maps describing each of the involved OpenShift objects:

```
def pod = [ apiVersion: "v1", kind: "Pod",
            metadata: [ name: "apod", labels: [ app: "myapp" ] ],
            spec: [ containers: [
                [ name: "mycontainer", image: "myproject/myimage:latest",
                  ports: [ [ name: "http", protocol: "TCP", containerPort: 8080 ] ] ] ]
          ] ]
def svc = [ apiVersion: "v1", kind: "Service",
            metadata: [ name: "asvc" ],
            spec: [ ports: [ [ port: 8080, targetPort: 8080 ] ],
                    selector: [ app: "myapp" ] ]
          ]
List<Map> list = new ArrayList<Map>()
list.add(pod)
list.add(svc)
def s = openshift.create(list, "--loglevel=5")
```

As mentioned above, all three forms also work with the `apply()` and `replace()` operations, according to their definition: `create()` will succeed if none of the resources exist and fail otherwise, `replace()` will only succeed if the resources already exist and fail if they are missing, whereas `apply()` will always work, creating resources if they are missing, or updating them if they already exist.

Using Selectors to Work with Resources

There are several forms of the `selector()` method:

```
openshift.selector(qualifiedName:String)
openshift.selector(kind:String, name:String)

openshift.selector(kind:String)
openshift.selector(kind:String, labels:Map<String, String>)
```

The first two return a so called static selector, where the qualified name produces a fixed list of resources (usually one), whereas the second two return a dynamic selector, which can change based on current project status. Before starting to work on a dynamic selector, it is recommended to use the `freeze()` method which will convert it into a static one.

Example use:

```
def s = openshift.selector(“bc/simple”)             (1)
def s = openshift.selector(“bc”, “simple”)          (1)

def d = openshift.selector(“pod”)                   (2)
def d = openshift.selector(“pod”, [ app: “simple” ])  (3)
```

1. In both cases, this selector refers to a single `BuildConfig` with the name of `simple`.
2. This dynamic selector returns a reference to all pods in the project.
3. The label filter will reduce the list of pods to only those that have a label of `app: simple`.

There are many methods that can be applied to a selector, such as:

Method	Description
`def s = d.freeze()`	Freezes the current dynamic selection of resources and converts it into a static selector.
`def u = s.union(x:Selector)`	Returns a static selector that is a union of selectors `s` and `x`.
`def x = s.narrow(t:String)`	Returns a new static selector that only points to resources of type `t` that were originally contained within `s`.
`def x = s.related(t:String)`	Returns a dynamic selector that only points to resources of type `t`, related to any resources contained within `s`.
`s.withEach { ... }`	Executes closure body once for each resource. When invoked with a dynamic selector, uses `freeze()` every time before the closure is executed.
`s.untilEach(m:Integer=1) { ... }`	If selector contains at least `m` objects, the closure is executed until closure body

Method	Description
	returns `true` for all the selected objects.
`def n = s.count()`	Returns an integer representing the number of resources in selector `s`.
`def n = s.name()`	Returns the name of the object selector `s` is pointing to. Error if there are multiple objects (in that case the `names()` method should be used).
`def n = s.names()`	Returns a list of qualified names (strings) belonging to selector `s`.
`def o = s.object()`	Returns a Groovy map containing unmarshalled JSON belonging to the object the selector is pointing to. Error if there are multiple objects.
`def o = s.objects()`	Returns a list of maps containing unmarshalled JSON belonging to objects that the selector is pointing to.
`s.describe([p...:String])`	Executes the `oc describe` operation on each resource contained in the selector. Any parameters are passed verbatim to the `oc` command.
`s.logs([p...:String])`	Executes the `oc logs` operation on each resource contained in the selector. Any parameters are passed verbatim to the `oc` command.
`s.label(m:Map[, p...:String])`	Applies labels contained in the map `m` to all the objects the selector is pointing to. Any additional parameters are passed verbatim to the `oc` command.

`s.scale([p...:String])`
`s.autoscale([p...:String])`
`s.volume([p...:String])`
	Executes the corresponding `oc` operation on each `DeploymentConfig` contained in the selector. Any parameters are passed verbatim to the `oc` command. Error if selector points to non-DC objects (use with `narrow()`).

`def r = s.rollout()`	With a `DeploymentConfig`, returns a `RolloutManager` object that can be used to further control that DC. See below.
`s.expose([p...:String])`	Exposes the `DeploymentConfig` or `Service` associated with the selector. Any parameters are passed verbatim to the `oc` command. Error if selector points to incompatible objects (use with `narrow()`).

`s.startBuild([p...:String])`
`s.cancelBuild([p...:String])`
	Executes the corresponding `oc` operation on each `BuildConfig` contained in the selector. Any parameters are passed verbatim to the `oc` command. Error if selector points to non-BC objects (use with `narrow()`).

Using the RolloutManager

When certain that a selector only contains `DeploymentConfig` resources (for example using the `narrow()` method), it is possible to invoke the `rollout()` method that returns an instance of `RolloutManager`. A `RolloutManager` can be used to invoke sub-verbs of the `oc rollout` command, such as:

Method	Description
`r.latest([p...:String])`	Deploys the latest available version of resources described by the selector.
`r.pause([p...:String])`	Pauses any ongoing deployments referred to by the selector.
`r.resume([p...:String])`	Resumes any deployments referred to by the selector that are currently paused.
`r.undo([p...:String])`	Rolls back to a previous version of a deployment.
`r.status([p...:String])`	Monitors the status of a pending deployment until it finishes, or report its status.
`r.history([p...:String])`	Obtains the available history of a given deployment.

Any parameters specified in the above list of commands will be passed verbatim to the corresponding `oc rollout` command.

Guided Exercise 2.2: Set up application artifacts for Jenkins manually

Exercise Overview

This exercise will have you prepare all the auxiliary resources required to run a Jenkins pipeline.

Outcomes

After the exercise, the project will have contained everything necessary to run a simple three-stage pipeline:

- An application image S2I build.
- A unit-test, "dummy", S2I build with no output.
- An additional ImageStreamTag to mark successfully tested builds as such.
- A deployment configuration for the microservice, which will only deploy tested images.

Exercise Resources/Configuration

Exercise Files Location:	`${HOME}/do388-manual-pipeline/`
Application URL:	`http://bonjour.apps.lab.example.com/api/bonjour`
Resources:	`N/A`

Before you begin...

Introductory notes as per usual (lab setup foo, etc.).

Steps

1. Create a new OpenShift project, call it "manual-pipeline".

```
[student@workstation ~]$ oc new-project manual-pipeline
Now using project "manual-pipeline" on server "https://master.lab.example.com:443".

You can add applications to this project with the 'new-app' command. For example, try:

    oc new-app centos/ruby-25-centos7~https://github.com/sclorg/ruby-ex.git

to build a new example application in Ruby.
```

2. Create a stand-alone build for the Bonjour microservice application image.

   2.1. Use the "oc new-build" command to create a new OpenShift build.

```
[student@workstation ~]$ oc new-build --name=bonjour-build --strategy=source \
> --image-stream=redhat-openjdk18-openshift:latest \
> -e MAVEN_MIRROR_URL=http://services.lab.example.com:8081/nexus/content/groups/\
> RedHatTraining \
> -e MAVEN_ARGS="-e -Popenshift -DskipTests package" \
> -e MAVEN_ARGS_APPEND="-pl bonjour" \
> -e ARTIFACT_DIR=bonjour/target \
> http://services.lab.example.com/hello-microservices.git
```

```
--> Found image c1bf724 (3 months old) in image stream "openshift/redhat-openjdk18-
openshift" under tag "latest" for "redhat-openjdk18-openshift:latest"
...
(output skipped)
...
--> Creating resources with label build=bonjour-build ...
    imagestream.image.openshift.io "bonjour-build" created
    buildconfig.build.openshift.io "bonjour-build" created
--> Success
```

2.2. Immediately cancel the build that is started automatically.

```
[student@workstation ~]$ oc cancel-build bc/bonjour-build
build.build.openshift.io/bonjour-build-1 marked for cancellation, waiting to be
cancelled
build.build.openshift.io/bonjour-build-1 cancelled
```

2.3. Make certain all the triggers on the build configuration are turned off.

```
[student@workstation ~]$ oc set triggers bc/bonjour-build --manual
buildconfig.build.openshift.io/bonjour-build triggers updated
```

2.4. Also, make the build incremental to save time when rebuilding.

```
[student@workstation ~]$ oc patch bc/bonjour-build \
> --patch='{ "spec": { "strategy": { "sourceStrategy": { "incremental": true }}}}'
buildconfig.build.openshift.io/bonjour-build patched
```

3. Create another build configuration, this time without output, simply to execute the unit tests.

3.1. Again, use the “oc new-build” command, but this time with “--no-output” option and slightly different MAVEN_ARGS which ensure that the unit tests are executed.

```
[student@workstation ~]$ oc new-build --name=bonjour-test --strategy=source \
> --no-output --image-stream=redhat-openjdk18-openshift:latest \
> -e MAVEN_MIRROR_URL=http://services.lab.example.com:8081/nexus/content/groups/\
> RedHatTraining \
> -e MAVEN_ARGS=”-e -Popenshift package” \
> -e MAVEN_ARGS_APPEND=”-pl bonjour” \
> -e ARTIFACT_DIR=bonjour/target \
> http://services.lab.example.com/hello—microservices.git
--> Found image c1bf724 (3 months old) in image stream "openshift/redhat-openjdk18-
openshift" under tag "latest" for "redhat-openjdk18-openshift:latest"
...
(output skipped)
...
--> Creating resources with label build=bonjour-test ...
    imagestream.image.openshift.io "bonjour-test" created
```

```
        buildconfig.build.openshift.io "bonjour-test" created
    --> Success
```

3.2. Again, immediately cancel the build that is started automatically.

```
[student@workstation ~]$ oc cancel-build bc/bonjour-test
build.build.openshift.io/bonjour-test-1 marked for cancellation, waiting to be
cancelled
build.build.openshift.io/bonjour-test-1 cancelled
```

3.3. Make certain all the triggers on the build configuration are turned off.

```
[student@workstation ~]$ oc set triggers bc/bonjour-test --manual
buildconfig.build.openshift.io/bonjour-test triggers updated
```

3.4. Noticing that the “oc new-build” command also created an image stream, although we specified this build will produce no output, remove the bogus image stream.

```
[student@workstation ~]$ oc delete is/bonjour-test
imagestream.image.openshift.io "bonjour-test" deleted
```

3.5. Note that we did not make the BuildConfig incremental in this case. Why?

4. Finally, create the deployment configuration for our microservice. We use the “--allow-missing-imagestream-tags” option because currently, we do not have a tagged image yet. Also, because there is no image, we are going to have to supply some missing information such as exposed ports, create a service, and finally, a route.

4.1. Create the deployment configuration using “oc new-app”.

```
[student@workstation ~]$ oc new-app --name=bonjour \
> --allow-missing-imagestream-tags --image-stream=bonjour-build:tested
--> Found tag :tested in image stream "manual-pipeline/bonjour-build" for "bonjour-
build:tested"

    * This image will be deployed in deployment config "bonjour"

--> Creating resources ...
    imagestreamtag.image.openshift.io "bonjour:tested" created
    deploymentconfig.apps.openshift.io "bonjour" created
--> Success
    Run 'oc status' to view your app.
```

4.2. Noticing again that the “oc new-app” command created an additional image stream, remove it.

```
[student@workstation ~]$ oc delete is/bonjour
imagestream.image.openshift.io "bonjour" deleted
```

4.3. Patch the deployment configuration, supplying the missing ports.

```
[student@workstation ~]$ oc patch dc/bonjour --patch='{ "spec": { "template": \
> { "spec": { "containers": [ { "name": "bonjour", "ports": [ \
> { "containerPort": 8080, "protocol": "TCP" }, \
> { "containerPort": 8443, "protocol": "TCP" }, \
> { "containerPort": 8778, "protocol": "TCP" } ] } ] }}}}'
deploymentconfig.apps.openshift.io/bonjour patched
```

4.4. Just like with the builds, make sure any triggers for this deployment configuration are turned off.

```
[student@workstation ~]$ oc set triggers dc/bonjour --manual
deploymentconfig.apps.openshift.io/bonjour triggers updated
```

4.5. Use this deployment configuration to expose a service, then expose that service as a route.

```
[student@workstation ~]$ oc expose dc/bonjour
service/bonjour exposed
[student@workstation ~]$ oc expose svc/bonjour --hostname=bonjour.apps.lab.example.com
route.route.openshift.io/bonjour exposed
```

5. The pom.xml file for the Bonjour application binds the Fabric8 Maven Plugin to phase "package". Remove that binding.

5.1. Clone the application to the workstation VM.

```
[student@workstation ~]$ git clone \
                           http://services.lab.example.com/hello-microservices.git
Cloning into 'hello-microservices'...
remote: Counting objects: 312, done.
remote: Compressing objects: 100% (174/174), done.
remote: Total 312 (delta 83), reused 248 (delta 53)
Receiving objects: 100% (312/312), 59.00 KiB | 8.43 MiB/s, done.
Resolving deltas: 100% (83/83), done.
```

5.2. Open the pom.xml file of the Bonjour microservice in an editor.

```
[student@workstation ~]$ cd hello-microservices/bonjour/
[student@workstation bonjour]$ vim pom.xml
```

5.3. Search for the following XML element (around line 85) and remove it or comment it out:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <phase>package</phase>
      <goals>
        <goal>resource</goal>
```

```
            <goal>build</goal>
          </goals>
      </execution>
```

5.4. Commit and push the source code back to the Git server.

```
[student@workstation bonjour]$ git commit -a -m 'unbound FMP from package phase'
[master 60f7f45] unbound FMP from package phase
 1 file changed, 1 insertion(+), 1 deletion(-)

[student@workstation bonjour]$ git push
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 752 bytes | 752.00 KiB/s, done.
Total 8 (delta 6), reused 0 (delta 0)
To http://services.lab.example.com/hello-microservices.git
   f9bd80c..e85c664  master -> master
```

This concludes this practice exercise - we have successfully set up everything needed to run a simple Jenkins pipeline.

Guided Exercise 2.3: Create and Run a Jenkins Pipeline

Exercise Overview

This exercise will have us finish a three-stage Jenkins pipeline which orchestrates the resources created in the previous exercise.

Outcomes

We will have written a JenkinsPipeline strategy BuildConfig that will:

- only run one build at a time, ignore any queued build requests and simply run the latest one
- monitor the ImageStreamTag redhat-openjdk18-openshift:latest for changes and automatically start a build
- have a GitHub and Generic webhook trigger that can start it

The pipeline itself will:

- be executed in a dedicated agent running a Maven-based image
- have a maximum total pipeline execution time of 15 minutes
- ask the user whether to run unit tests, but
- proceed and run the unit tests anyway if no response is received within 60 seconds
- start the bonjour application build
- start the bonjour application unit tests, if so desired
- wait for both of them to finish, but only up to 10 minutes
- if a timeout occurs during the wait, terminate both builds and abort the pipeline
- if either of the builds was unsuccessful, terminate the other build and abort the pipeline, otherwise
- tag the current image as tested, and
- deploy the latest version of the application and wait for it to finish
- if the deployment fails, try it again up to three times total

The pipeline code will also report the outcome at the end of pipeline execution, including the name of the stage where the build failed or got interrupted, if that is what happened.

Exercise Resources/Configuration

Exercise Files Location:	`${HOME}/do388-manual-pipeline/`
Application URL:	`http://bonjour.apps.lab.example.com/api/bonjour`
Resources:	`manual-pipeline.yml`

Before you begin...

Introductory notes as per usual (lab setup foo, etc.).

Steps

1. Create a new file and code the pipeline definition in it.

   1.1. In your favorite editor, open a file called `bonjour-pipeline.yml`.

1.2. First, let's create the `BuildConfig` YAML outline:

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: bonjour-pipeline
spec:
  runPolicy: SerialLatestOnly                                 (I)
  source:
    type: None
  strategy:
    successfulBuildsHistoryLimit: 3
    failedBuildsHistoryLimit: 2
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        PIPELINE-GOES-HERE                                    (II)
    type: JenkinsPipeline
  triggers:
  - type: ImageChange                                         (III)
    imageChange:
      from:
        kind: ImageStreamTag
        name: redhat-openjdk18-openshift:latest
        namespace: openshift
  - type: GitHub                                              (IV)
    github:
      secret: 3Uj16Hmi7dsAABxG0ZHgeBWZ30SpGUJ6
  - type: Generic                                             (IV)
    generic:
      secret: 8gF5xpJ9VIBhTpcvUtu8vrPJlm8dIlTv
  - type: ConfigChange                                        (V)
```

I. this ensures that only one build is run at a time, ignoring any queued build requests
II. we will be implementing the pipeline inline here in the upcoming steps of this exercise
III. the image change trigger will monitor the `redhat-openjdk18-openshift:latest` for changes
IV. the webhooks allow us to start the pipeline whenever code is pushed to relevant source code repositories
V. configuration change triggers in build configs are ignored for now, but let's include one for future reference
NOTE: you can generate the two secrets using a command like "`pwgen -cns 32 2`"

1.3. Next, create a basic outline of the pipeline. In your existing pipeline, replace the text saying

```
PIPELINE-GOES-HERE
```

with the following skeleton of the pipeline-to-be:

```
pipeline {
  agent { node { label "maven" } }
  environment {
  }
  options {
  }
  stages {
    stage("Ask For User Input") {
    }
    stage("Start Image Build") {
    }
    stage("Start Unit Tests") {
    }
    stage("Wait For Builds") {
    }
    stage("Tag New Image") {
    }
    stage("Deploy Application") {
    }
  }
  post {
    success {
    }
    failure {
    }
    aborted {
    }
    cleanup {
    }
  }
}
```

Note the `agent` keyword which makes sure this pipeline will execute in a dedicated pod running the Jenkins Maven agent. We will add code to the other sections and stages as we go along.

1.4. Define environment variables, and add execution options. At the beginning of the pipeline, add the following:

```
pipeline {
  agent { node { label "maven" } }
  environment {
    RUN_TESTS = true
    STAGE = null
    BLD_IMAGE = null
    BLD_TESTS = null
  }
  options {
    timeout(time: 15, unit: "MINUTES")
  }
```

The above four variables allow us to remember whether to run the unit tests, which stage was executed last in the case of a failure, and which names have been given to the two builds we start (so we can access them later). We will add code to initialize the `STAGE` variable at the beginning of each stage in the next step. The `timeout` option takes care of the 15 minute time limit for the entire pipeline.

1.5. Add the stage name declarations to each stage (remember that assignments to variables are only allowed in a `script` closure within a declarative pipeline):

```
stages {
  stage("Ask For User Input") {
    steps {
      script {
        STAGE = "input"
      }
    }
  }
  stage("Start Image Build") {
    steps {
      script {
        STAGE = "build"
      }
    }
  }
  stage("Start Unit Tests") {
    steps {
      script {
        STAGE = "test"
      }
    }
  }
  stage("Wait For Builds") {
    steps {
      script {
        STAGE = "wait"
      }
    }
  }
  stage("Tag New Image") {
    steps {
      script {
        STAGE = "tag"
      }
    }
  }
  stage("Deploy Application") {
    steps {
      script {
        STAGE = "deploy"
      }
    }
    }
  }
```

1.6. Make the unit test stage conditional by adding a `when` closure to it, testing for the value of `RUN_TESTS` variable:

```
stage("Start Unit Tests") {
  when {
    expression { return RUN_TESTS }
  }
...
```

1.7. Mark the deployment stage as being retried up to three times:

```
stage("Deploy Application") {
  options {
    retry(3)
  }
...
```

1.8. Add the post-execution reports. In the `post` section of the pipeline, add a couple of `echo` statements:

```
post {
  success {
    echo "Pipeline completed successfully."
  }
  failure {
    echo "ERROR: Pipeline failed in stage \"${STAGE}\"."
  }
  aborted {
    echo "ERROR: Pipeline aborted in stage \"${STAGE}\"."
  }
  cleanup {
    echo "Shutting down after ${currentBuild.durationString}, " +
         "with result of ${currentBuild.currentResult}"
  }
}
```

Note the use of variables `STAGE` and (the implicit) `currentBuild`.

2. Implement the various stages.

2.1. Ask the user whether to run the unit tests or not, but only wait up to 60 seconds for feedback:

```
stage("Ask For User Input") {
  steps {
    script {
      STAGE = "input"
      try {
        timeout(time: 60, unit: "SECONDS") {
          def response = input(
                message: "Please provide input and click 'Proceed' in 60 seconds.",
                parameters: [ booleanParam(name: "run_tests",
                              description: "Run unit tests?",
                              defaultValue: true)])
          RUN_TESTS = response
        }
      } catch (exc) {
        echo "Stage \"${STAGE}\" timed out or interrupted: ${exc}; ignoring."
      }
    }
  }
}
```

Note that both the `timeout` and the `input` directives above are wrapped in a `try/catch/finally` block. This ensures that the pipeline will continue execution regardless of whether the user provides input within the allocated time, lets this step time out, or clicks the `Abort` button.

2.2. Start the `bonjour` image build in the next stage:

```
stage("Start Image Build") {
  steps {
    script {
      STAGE = "build"
      openshift.withCluster() { openshift.withProject() {
        BLD_IMAGE = openshift.selector("bc/bonjour-build")
                      .startBuild().object().metadata.name
        echo "Build started successfully as ${BLD_IMAGE}."
      }}
    }
  }
}
```

Notice how we store the name of the build in `BLD_IMAGE` variable. We will use that in the `wait` stage.

2.3. Similarly to image build, start the unit tests:

```
stage("Start Unit Tests") {
  when {
    expression { return RUN_TESTS }
  }
  steps {
    script {
      STAGE = "test"
      openshift.withCluster() { openshift.withProject() {
        BLD_TESTS = openshift.selector("bc/bonjour-test")
                      .startBuild().object().metadata.name
        echo "Tests started successfully as ${BLD_TESTS}."
      }}
    }
```

```
      }
    }
```

2.4. Wait for both builds to complete and check their outcomes:

```
    stage("Wait For Builds") {
      steps {
        script {
          STAGE = "wait"
          openshift.withCluster() { openshift.withProject() {
            def bi = openshift.selector("builds/${BLD_IMAGE}")          (I)
            def bt = null
            if (BLD_TESTS != "null") {
              openshift.selector("builds/${BLD_TESTS}")                 (I)
            }
            try {
              timeout(time: 10, units: "MINUTES") {                     (II)
                waitUntil {                                             (III)
                  if (bi.object().status.phase == "Failed" ||           (IV)
                      bi.object().status.phase == "Cancelled" ||
                      (bt != null &&
                        (bt.object().status.phase == "Failed" ||
                        bt.object().status.phase == "Cancelled"))) {
                    throw new RuntimeException("Builds failed.")
                  }
                  return bi.object().status.phase == "Complete" &&      (V)
                         (bt == null ||
                         (bt != null &&
                          bt.object().status.phase == "Complete"))
                }
              }
            } catch (exc) {                                             (VI)
              if (bi.object().status.phase == "Running") {              (VII)
                echo "Terminating build ${BLD_IMAGE}."
                bi.cancelBuild()
              }
              if (bt != null bt.object().status.phase == "Running") {   (VII)
                echo "Terminating build ${BLD_TESTS}."
                bt.cancelBuild()
              }
              error("Build(s) failed: " +                               (VIII)
                    "${BLD_IMAGE} = ${bi.object().status.phase}, " +
                    "reason ${bi.object().status.reason}, " +
                    (bt != null ?
                      "${BLD_TESTS} = ${bt.object().status.phase}, " +
                      "reason ${bt.object().status.reason}" :
                      "unit tests skipped"))
            }
        }}
      }
    }
  }
```

I. using the two pipeline-scoped variables, `BLD_IMAGE` and `BLD_TESTS` to obtain references to the two builds; do mind that the unit tests may have been skipped altogether, which complicates the rest of this stage a bit
II. set a timeout for both builds to finish in 10 minutes
III. run the `waitUntil` closure until both builds complete (also see V)
IV. if either of the builds has failed or was cancelled, throw an exception to terminate `waitUntil` (see also VI)
V. if both builds have completed, this expression will evaluate to `true` and thus end `waitUntil` successfully
VI. the `try/catch` block is positioned such that it encloses both the `timeout` and the build failures
VII. if either of the builds is still running while the exception occurs, they are terminated
VIII. throw a new `error` with a custom message, telling the user what exactly failed and why

2.5. Tag the `latest` image as `tested` if we made it so far:

```
stage("Tag New Image") {
  steps {
    script {
      STAGE = "tag"
      openshift.withCluster() { openshift.withProject() {
        openshift.tag("bonjour-build:latest", "bonjour-build:tested")
      }}
    }
  }
}
```

2.6. Finally, start the application rollout and wait for it to finish deploying:

```
stage("Deploy Application") {
  options {
    retry(3)
  }
  steps {
    script {
      STAGE = "deploy"
      openshift.withCluster() { openshift.withProject() {
        def dc = openshift.selector("dc/bonjour")
        dc.rollout().latest()
        dc.rollout().status("-w")
      }}
    }
  }
}
```

Notice how in this stage, as opposed to constantly polling for status (as we did in the `build` stage), we simply let the `RolloutManager` wait for the outcome of the `status()` operation - it will propagate its outcome to the pipeline.

3. Create the pipeline in the OpenShift project.

3.1. Send the YAML definition to OpenShift:

```
[student@workstation ~]$ oc create -f bonjour-pipeline.yml
buildconfig.build.openshift.io/bonjour-pipeline created
```

3.2. Wait for the Jenkins pod to deploy and fully start up:

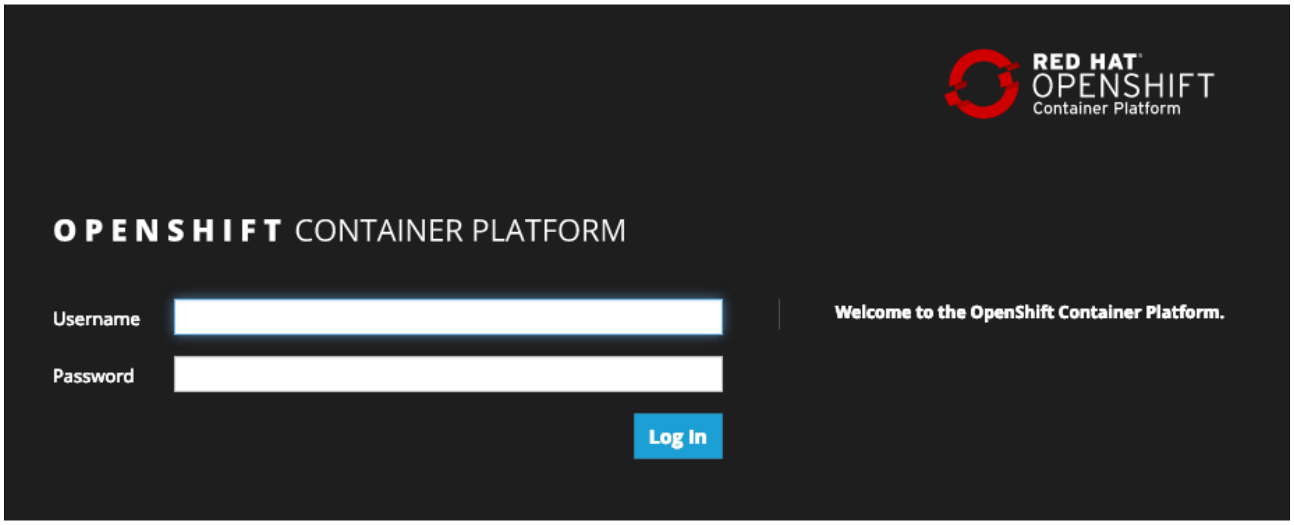
```
[student@workstation ~]$ oc get pods -w
NAME                READY     STATUS    RESTARTS   AGE
jenkins-1-deploy    1/1       Running   0          10s
jenkins-1-tjz9c     0/1       Running   0          8s
jenkins-1-tjz9c     1/1       Running   0          39s
jenkins-1-deploy    0/1       Completed   0        42s
jenkins-1-deploy    0/1       Terminating   0      42s
```

3.3. Notice that the build triggered automatically due to an ImageChangeTrigger:

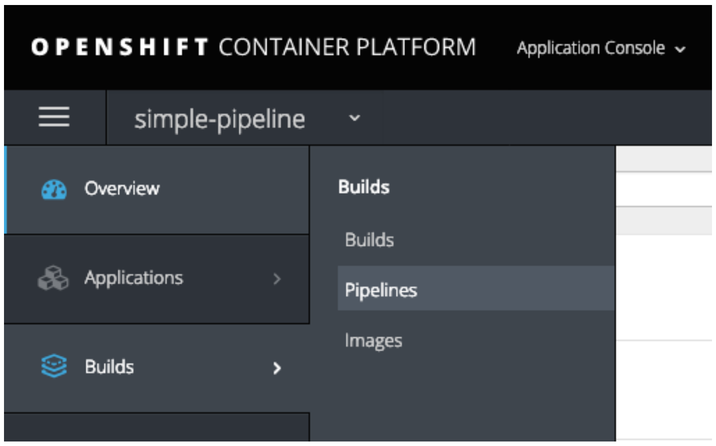
```
[student@workstation ~]$ $ oc get builds
NAME               TYPE            FROM         STATUS        STARTED          DURATION
bonjour-build-1    Source          Git@2fb5174 Cancelled...  5 minutes ago    7s
bonjour-test-1     Source          Git@2fb5174 Cancelled...  4 minutes ago    6s
bonjour-pipeline-1 JenkinsPipeline             Running      12 seconds ago

[student@workstation ~]$ oc describe build bonjour-pipeline-1
Name:          bonjour-pipeline-1
...
(output omitted)
...
Build trigger cause: Image change
Image ID:            docker-registry.default.svc:5000/openshift/redhat-openjdk18-
openshift@sha256:8d2fda5e9f2820f2df99968efa99799f8ea7abb3f74586a75246e86491bec8d7
Image Name/Kind:     redhat-openjdk18-openshift:latest / ImageStreamTag

Events:  <none>
```

4. Have the pipeline run with defaults once.

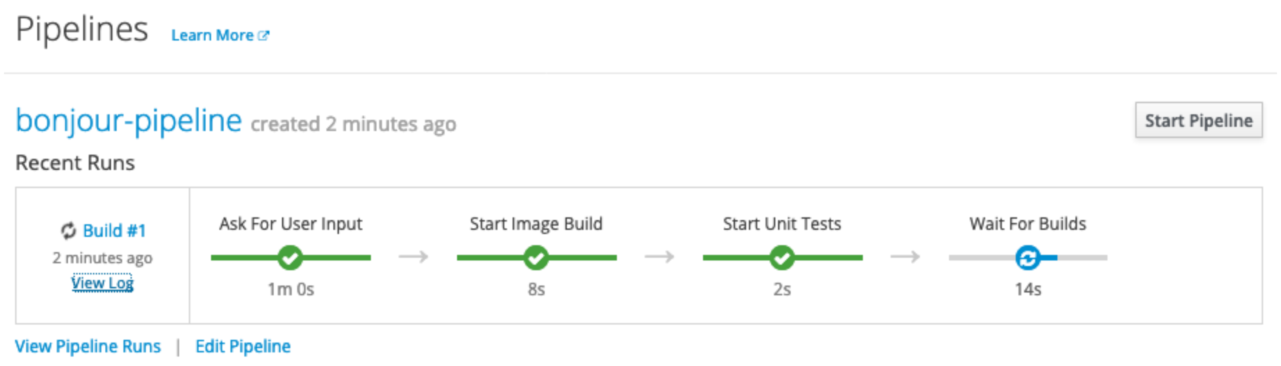
4.1. Log into the OpenShift Application Console:


RED HAT OPENSHIFT Container Platform
OPENSHIFT CONTAINER PLATFORM
Username
Password
Log In
Welcome to the OpenShift Container Platform.

4.2. Navigate to Builds/Pipelines:



4.3. Wait for the input prompt to time out after one minute and see that the unit tests are started:

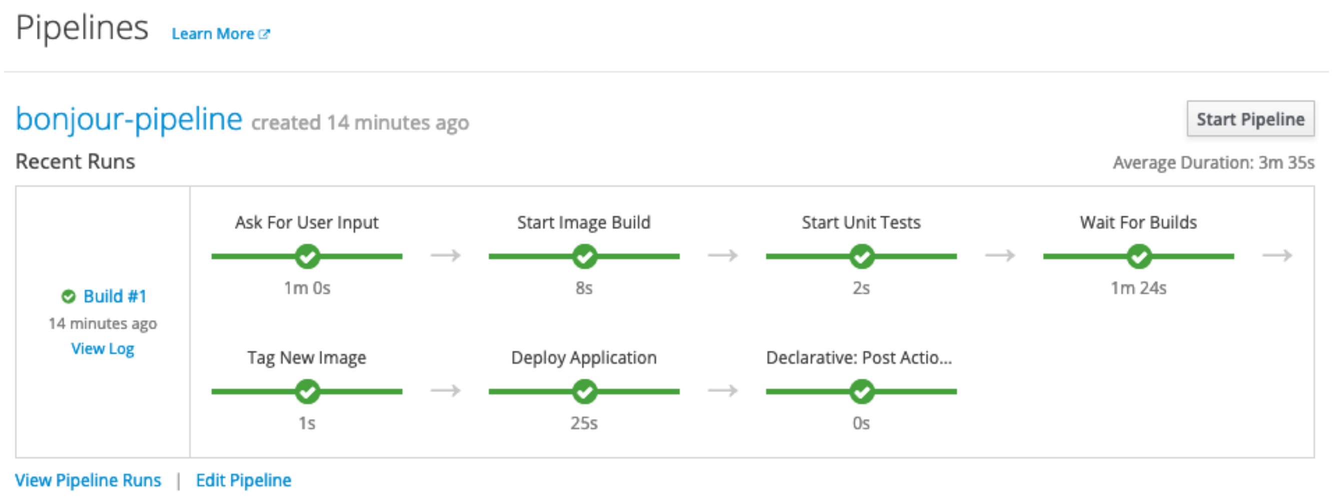


4.4. Have a look at the list of running pods in the terminal:

```
[student@workstation ~]$ oc get pods
NAME                   READY    STATUS    RESTARTS   AGE
bonjour-build-2-build  1/1      Running   0          14s
```

```
bonjour-test-2-build    1/1       Running   0          11s
jenkins-1-tjz9c         1/1       Running   0          4m
maven-cpwbx             1/1       Running   0          1m
```

4.5. Wait for the remainder of the pipeline to finish:



4.6. Click "View Log", log into Jenkins, and examine the pipeline logs:

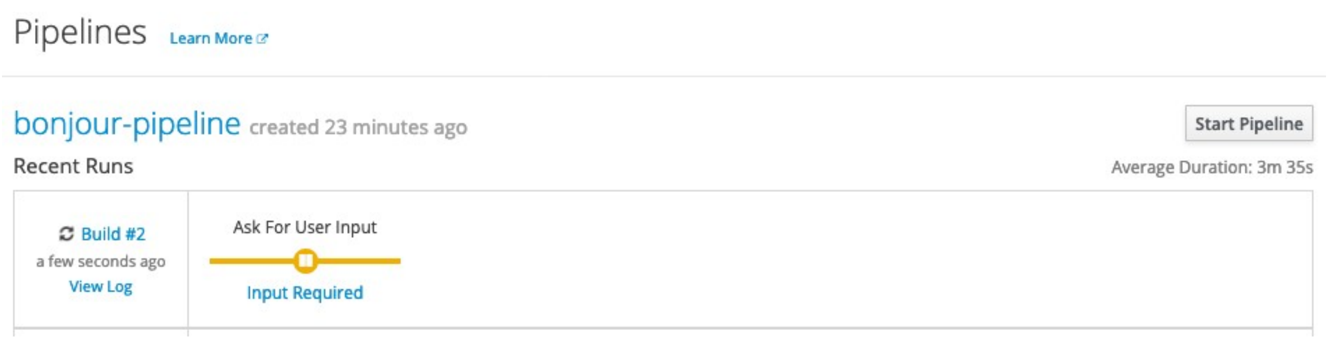
```
OpenShift Build manual-pipeline/bonjour-pipeline-1
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Still waiting to schedule task
'Jenkins' doesn't have label 'maven'
Agent maven-6jw0d is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (maven):
* [jnlp] infra.ocp.p0f.local/openshift3/jenkins-agent-maven-35-rhel7:latest

Running on maven-6jw0d in /tmp/workspace/manual-pipeline/manual-pipeline-bonjour-
pipeline
[Pipeline] {
[Pipeline] withEnv
[Pipeline] {
[Pipeline] timeout
Timeout set to expire in 15 min
...
(output omitted)
...
Timeout set to expire in 1 min 0 sec
[Pipeline] {
[Pipeline] input
Input requested
Cancelling nested steps due to timeout
[Pipeline] }
[Pipeline] // timeout
[Pipeline] echo
Stage "input" timed out or interrupted:
org.jenkinsci.plugins.workflow.steps.FlowInterruptedException; ignoring.
...
(output omitted)
...
Pipeline completed successfully.
[Pipeline] echo
Shutting down after 3 min 33 sec and counting, with result of SUCCESS
[Pipeline] }
...
(output omitted)
...
[Pipeline] End of Pipeline
Finished: SUCCESS
```

5. Re-run the pipeline, skipping the unit tests this time.

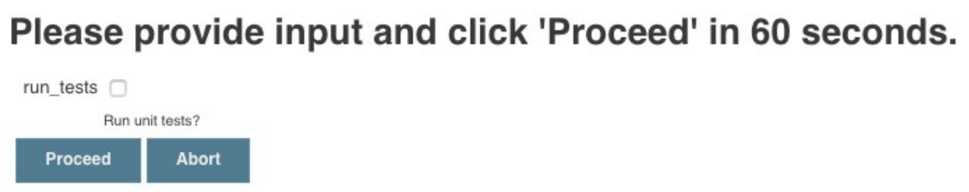
5.1. Start the bonjour-pipeline build again:

```
[student@workstation ~]$ $ oc start-build bonjour-pipeline
build.build.openshift.io/bonjour-pipeline-2 started
```

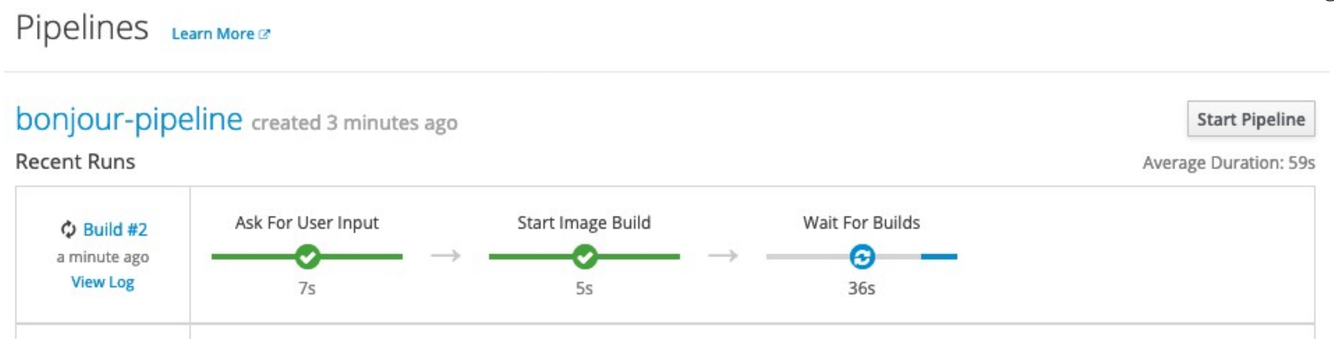
5.2. When the “Input Required” prompt appears, click on the link:



5.3. In the form displayed by Jenkins, uncheck the mark next to `run_tests` field and click “Proceed”:


Please provide input and click 'Proceed' in 60 seconds.

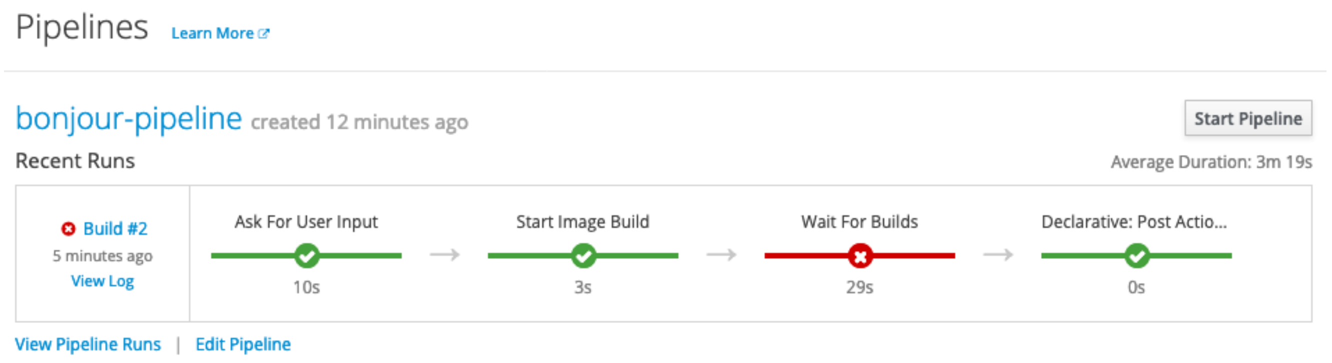
5.4. Back in the OpenShift Application Console, make sure the unit tests are skipped:



5.5. While the `bonjour-build` build is still running, cancel it:

```
[student@workstation ~]$ oc get builds -l build=bonjour-build
NAME            TYPE   FROM          STATUS       STARTED          DURATION
bonjour-build-1 Source Git           Cancelled... 18 minutes ago   2s
bonjour-build-2 Source Git@2fb5174   Complete     5 minutes ago    1m23s
bonjour-build-3 Source Git@2fb5174   Running      18 seconds ago

[student@workstation ~]$ oc cancel-build bonjour-build-3
build.build.openshift.io/bonjour-build-3 marked for cancellation, waiting to be
cancelled
build.build.openshift.io/bonjour-build-3 cancelled
```

5.6. Make sure the pipeline is marked as failed:



5.7. Click “View Log” and make sure the result is correctly reported in the Jenkins console:

```
[Pipeline] stage
[Pipeline] { (Tag New Image)
Stage "Tag New Image" skipped due to earlier failure(s)
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy Application)
Stage "Deploy Application" skipped due to earlier failure(s)
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
```

```
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
ERROR: Pipeline failed in stage "wait".
[Pipeline] echo
Shutting down after 1 min 9 sec and counting, with result of FAILURE
[Pipeline] }
...
(output skipped)
...
[Pipeline] End of Pipeline
ERROR: Build(s) failed: bonjour-build-3 = Cancelled, reason CancelledBuild, unit tests
skipped
Finished: FAILURE
```

5.8. Clean up:

```
[student@workstation ~]$ oc delete project manual-pipeline
project.project.openshift.io "manual-pipeline" deleted
```

This concludes the practice exercise. You have successfully implemented a Jenkins pipeline!