



# Continuous Integration/Continuous Deployment (CI/CD)

6.3.1

## Introduction to CI/CD



Continuous Integration/Continuous Deployment (CI/CD) is a philosophy for software deployment that figures prominently in the field of DevOps. DevOps itself is about communication and making certain that all members of the team are working together to ensure smooth operation.

6.3.2

## Continuous Integration



Have you ever done a lot of work on an application and when you tried to merge it back into the main application, there were many merge conflicts, any one of which carried the potential to introduce major bugs? Continuous Integration is intended to eliminate this problem.

The idea behind Continuous Integration is that you, and all other developers on the project, continually merge your changes with the main branch of the existing application. This means that any given change set is small and the potential for problems is low. If everyone is using the main branch, anyone who checks out code is going to have the latest version of what everyone else is developing.

As part of this process, developers are expected to perform extensive, and usually automated, testing on their code before merging back into the main branch. Doing this, the idea is that most issues are caught before they become a more serious problem.

The Continuous Integration process provides a number of additional benefits, as every commit provides an opportunity for the system to perform additional tasks. For example, the pipeline might be set up to perform these tasks:

- Code compilation
- Unit test execution
- Static code analysis
- Integration testing

- Packaging and versioning
- Publishing the version package to Docker Hub or other package repositories

Note that there are some situations that involve large and complicated changes, such as the addition of new features, in which changes must be grouped together. In this case, every commit may trigger only part of the CI pipeline, with the packaging and versioning steps running only when the entire feature is merged to the master.

In some cases, adjusting to this way of working requires a change in thinking on the part of the organization, or on the part of individual developers who may be used to working in their own branch, or on feature branches. This change is necessary, however, if you are going to achieve the next step: Continuous Delivery, which is not quite the same as Continuous Deployment.

## Continuous Delivery

Continuous Delivery is the process of developing in sprints that are short enough so that the code is always in a deployable state. With Continuous Integration, small change sets are continuously integrated into the main code branch. Continuous Delivery means that those changes are engineered to be self-contained to the point where at any given time, you could deploy a working application.

The process looks something like this:

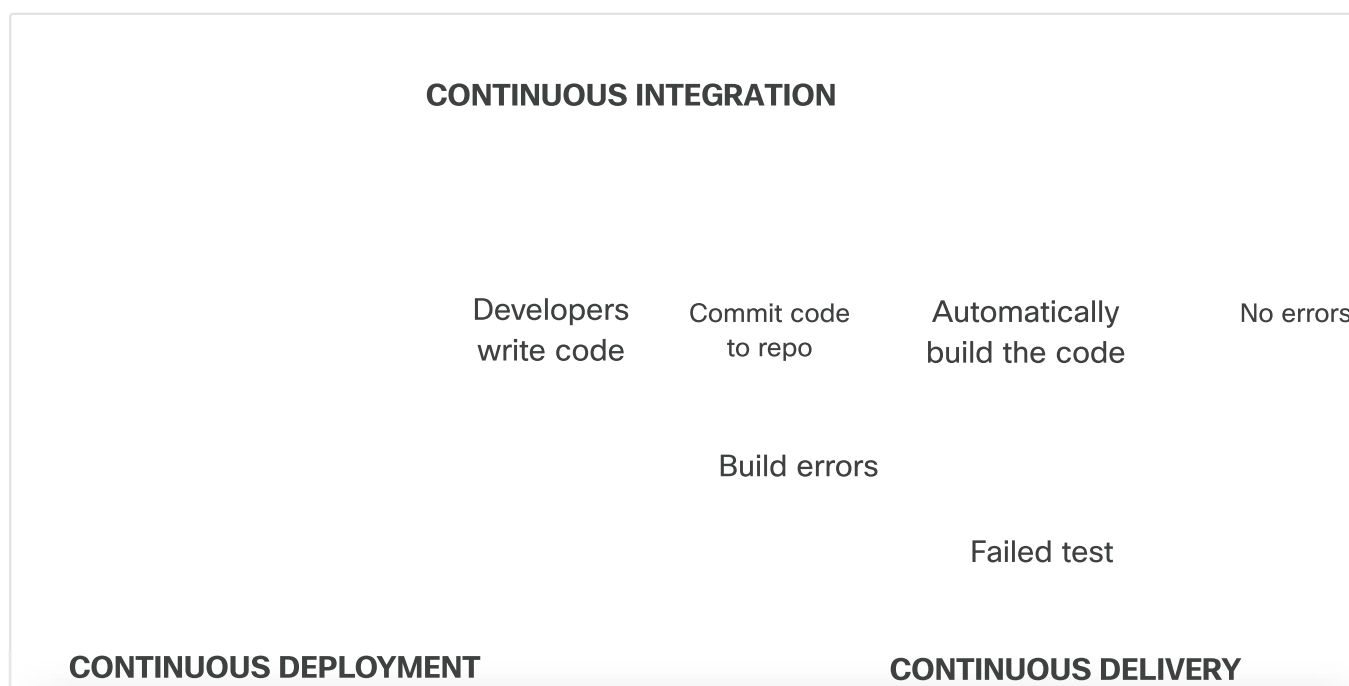
**Step 1.** Start with the version artifact that was created as part of the Continuous Integration process.

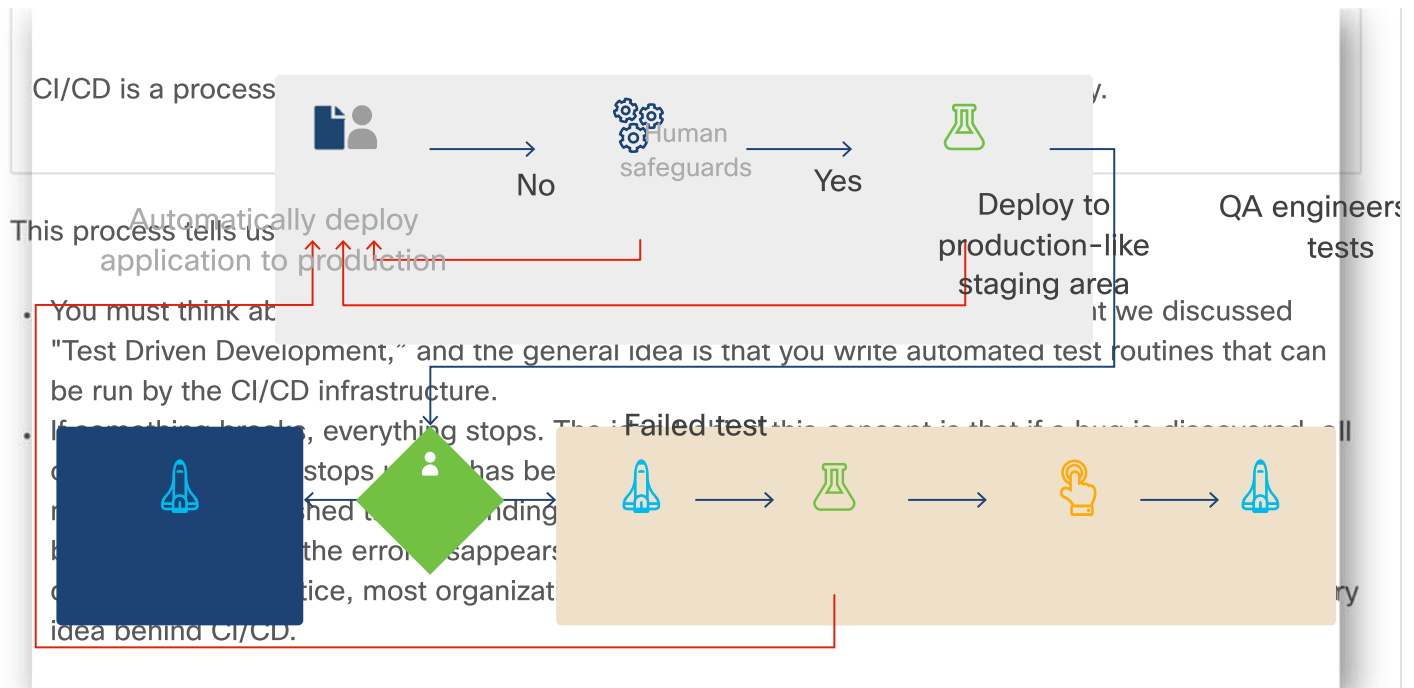
**Step 2.** Automatically deploy the candidate version on staging.

**Step 3.** Run integration tests, security tests, performance tests, scale tests, or other tests identified by the team or organization. These are known as gating tests because they determine whether this version of the software can be promoted further in the deployment process.

**Step 4.** If all gating tests pass, tag this build as suitable for production.

Note that Continuous Delivery does not mean you deploy constantly, that process is called Continuous Deployment. Continuous Delivery ensures that you always have a version that you *can* deploy.





## Continuous Deployment

Continuous Deployment is the ultimate expression of CI/CD. When changes are made, tested, integrated with the main branch, and tested again, they are deployed to production using automation. This means that code is being deployed to production constantly, which means your users are going to be your final testers. In other words, Continuous Deployment is a special type of Continuous Delivery, in which every build that is marked as ready for production gets deployed.

Some organizations favor this type of deployment because it means that users always have the most up to date code. Most organizations take a more cautious approach that requires a human to push the code to production.

## Preventing impact to users

Although we try to do extensive testing as part of the CI/CD process, there is always the possibility that a bad build will pass the gate. In order to avoid impacting users, or at least to limit the impact, you can use deployment strategies such as:

- **Rolling upgrade** - This is the most straightforward version of Continuous Delivery, in which changes are periodically rolled out in such a way that they do not impact current users, and nobody should have to "reinstall" the software.
- **Canary pipeline** - In this case, the new version is rolled out to a subset of users (or servers, depending on the architecture). If these users experience problems, the changes can be easily rolled back. If these users do not experience problems, the changes are rolled out to the rest of production.
- **Blue-green deployment** - In this case, an entirely new environment (Blue) is created with the new code on it, but the old environment (Green) is held in reserve. If users on the new environment experience problems, traffic can be diverted back to the original environment. If there are no problems within a specific amount of time, the new environment becomes the production environment and the old one is retired to be used for the next change.

## 6.3.3

## CI/CD Benefits



Companies are willing to make such a seemingly “drastic” change in their processes because of the benefits that come with using CI/CD for development. These benefits include:

- **Integration with agile methodologies** - Agile development is built around the idea of short sprints, after which the developer team delivers a functional application with some subset of the required features. CI/CD works within that same short sprint framework. Every commit is a version of the “deliver a working version of the software” concept.
- **Shorter Mean Time To Resolution (MTTR)** - Because change sets are small, it becomes much easier to isolate faults when they do occur, and to either fix them or roll them back and resolve any issues.
- **Automated deployment** - With automated testing and predictable deployment comes the ability to do automated deployments. This means it is possible to use deployment strategies such as canary release pipeline deployment, in which one set of users gets the new feature set and the rest gets the old. This process enables you to get live testing of the new feature to ensure it is functioning as expected before rolling it out to the entire user base.
- **Less disruptive feature releases** - With development proceeding in small chunks that always result in a deployable artifact, it is possible to present users with incremental changes rather than large-scale changes that can be disorienting to users.
- **Improved quality** - All of these benefits add up to higher quality software because it has been thoroughly tested before wide scale adoption. And because error resolution is easier, it is more likely to be handled in a timely manner rather than accruing technical debt.
- **Improved time to market** - Because features can be rolled out individually, they can be offered to users much more quickly than if they had to be deployed all at the same time.

## 6.3.4

## Example Build Job for Jenkins



**Note:** The steps shown in this rest of this topic are for instruction purposes only. Additional details that you would need to complete these commands in your DEVASC VM are not provided. However, you will complete similar steps in the lab *Build a CI/CD Pipeline Using Jenkins* later in the topic.

In this part we show a deployment pipeline, which is normally created with a build tool such as Jenkins. These pipelines can handle tasks such as gathering and compiling source code, testing, and compiling artifacts such as tar files or other packages. All these examples show screenshots from an existing Jenkins server.

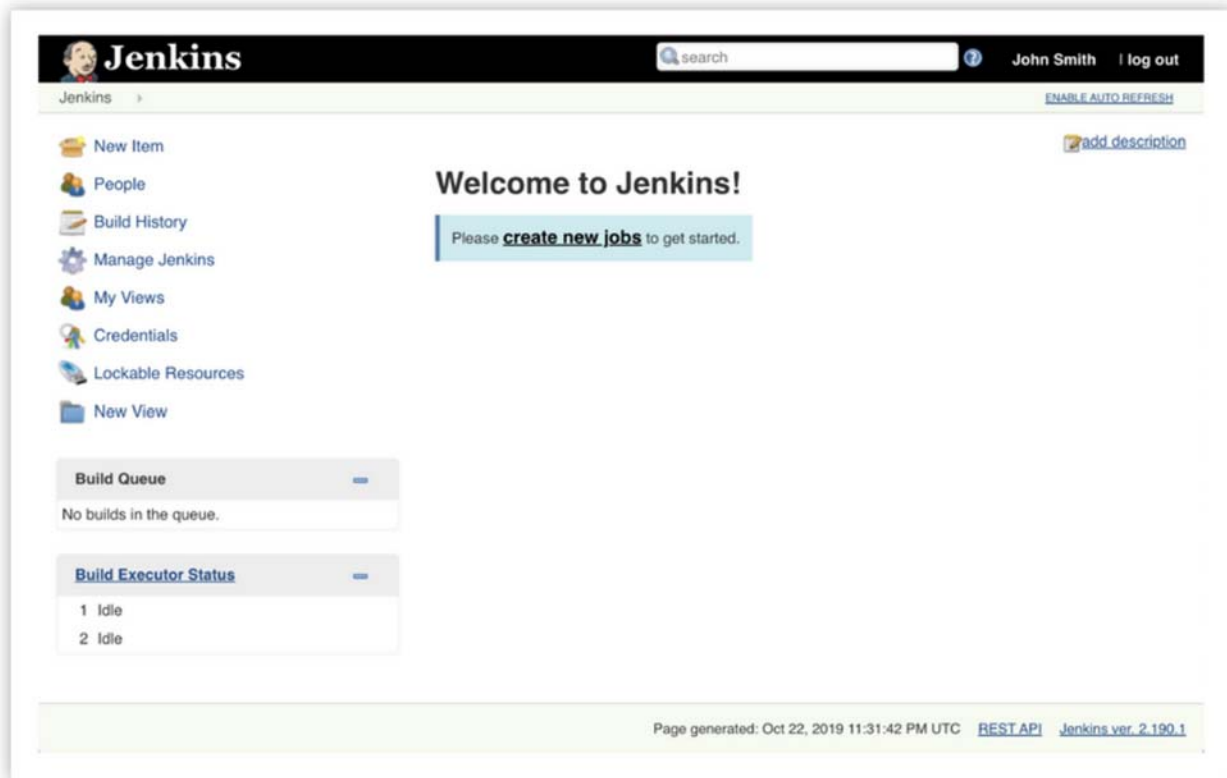
### Example build job for Jenkins

The fundamental unit of Jenkins is the project, also known as the job. You can create jobs that do all sorts of things, from retrieving code from a source code management repo such as GitHub, to building

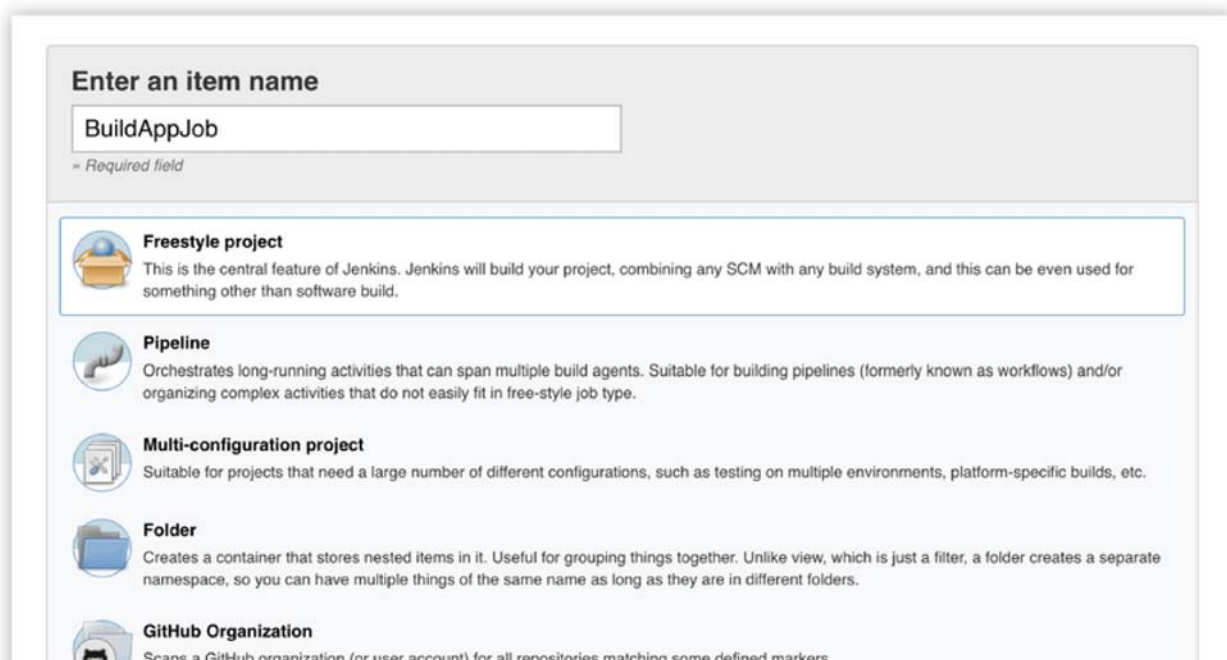
an application using a script or build tool, to packaging it up and running it on a server.

Here is a simple job that retrieves a version of the sample application from GitHub and runs the build script. Then you do a second job that tests the build to ensure that it is working properly.

First, create a New Item in the Jenkins interface by clicking the "create new jobs" link on the welcome page:

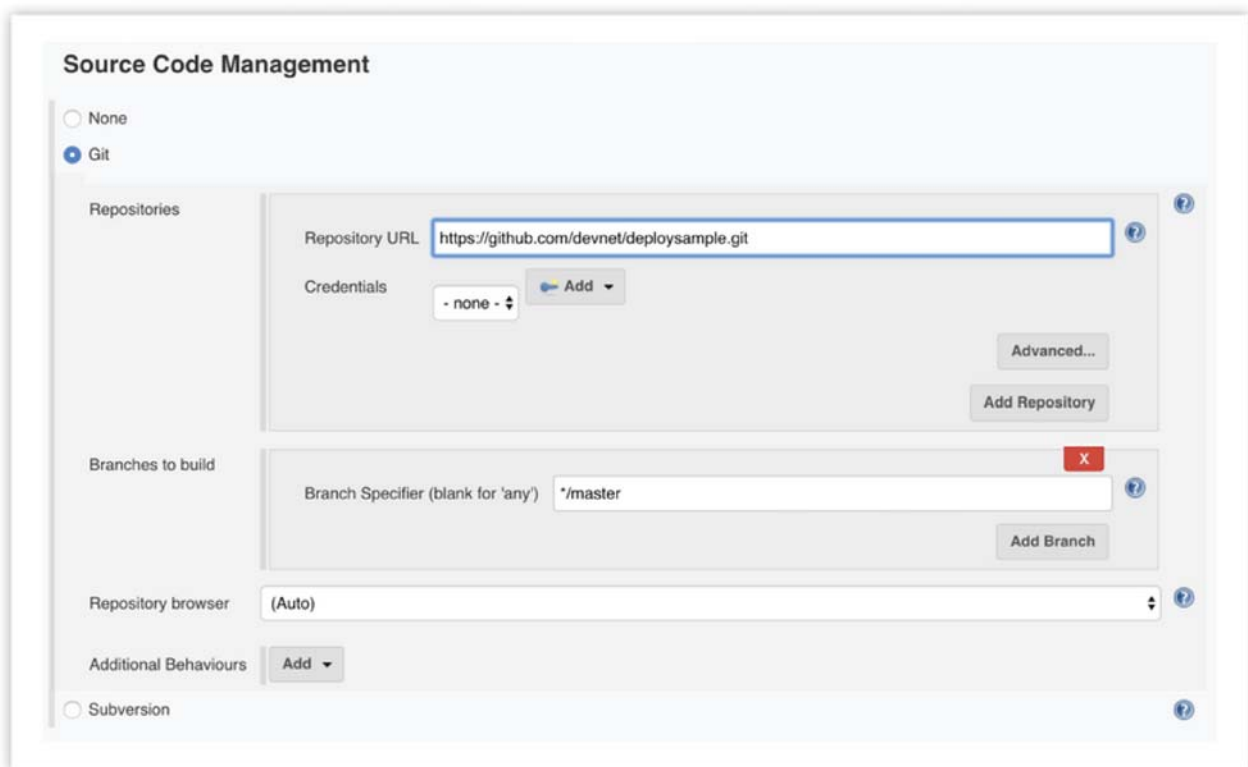


Enter a name, choose Freestyle project (so that you have the most flexibility) and click OK.

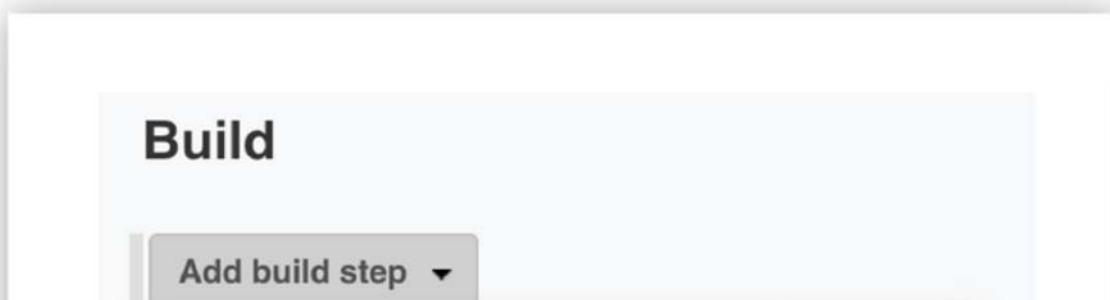


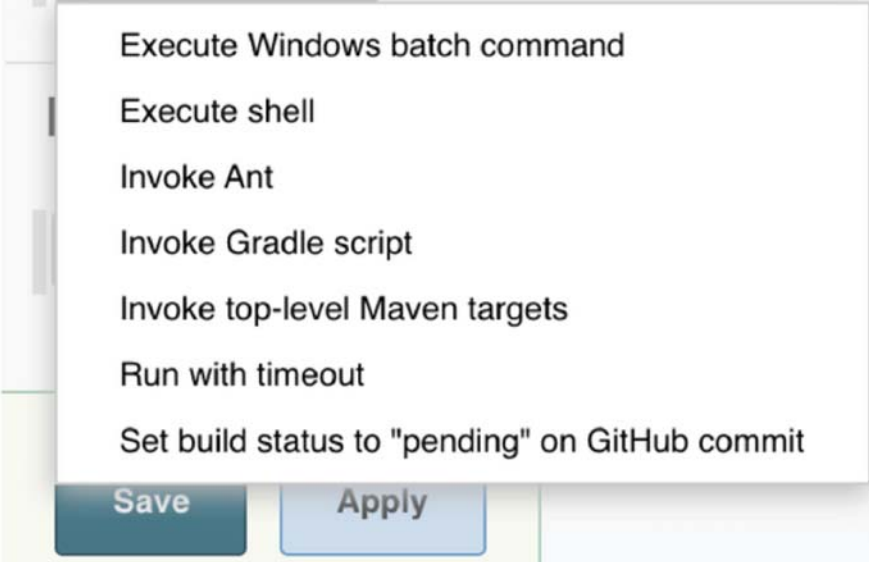


Scroll down to Source Code Management and select Git, then enter a GitHub repository URL for the Repository URL. Typically this is a repository to which you have write access so you can get scripts merged to. It is best to store build scripts within the repository itself and use version control on the script.



Now scroll down to Build and click Add Build Step. Choose Execute shell.





A screenshot of a dropdown menu with a white background and a light gray border. The menu is open, showing a list of build actions. At the bottom of the menu are two buttons: 'Save' in a dark blue box and 'Apply' in a light blue box. The background of the page is a light gray.

- Execute Windows batch command
- Execute shell
- Invoke Ant
- Invoke Gradle script
- Invoke top-level Maven targets
- Run with timeout
- Set build status to "pending" on GitHub commit

Save

Apply

In the Command box, add the command:

```
./buildscript.sh
```

This script is intended to be part of a repo, and is downloaded as a first step. From here you could use a post-build action to run another job, but you would have to create it first. Click Save to move on.

## Build

### Execute shell

Command `./buildscript.sh`

See [the list of available environment variables](#)

Advanced...

Add build step ▾

## Post-build Actions

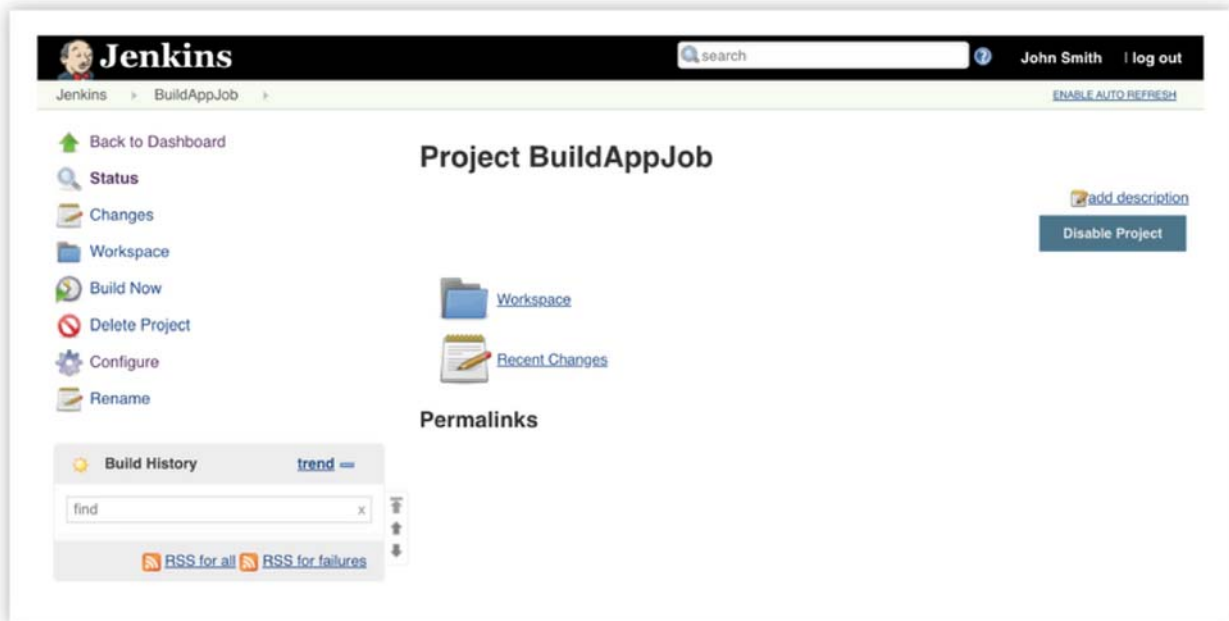
Add post-build action ▾

Save

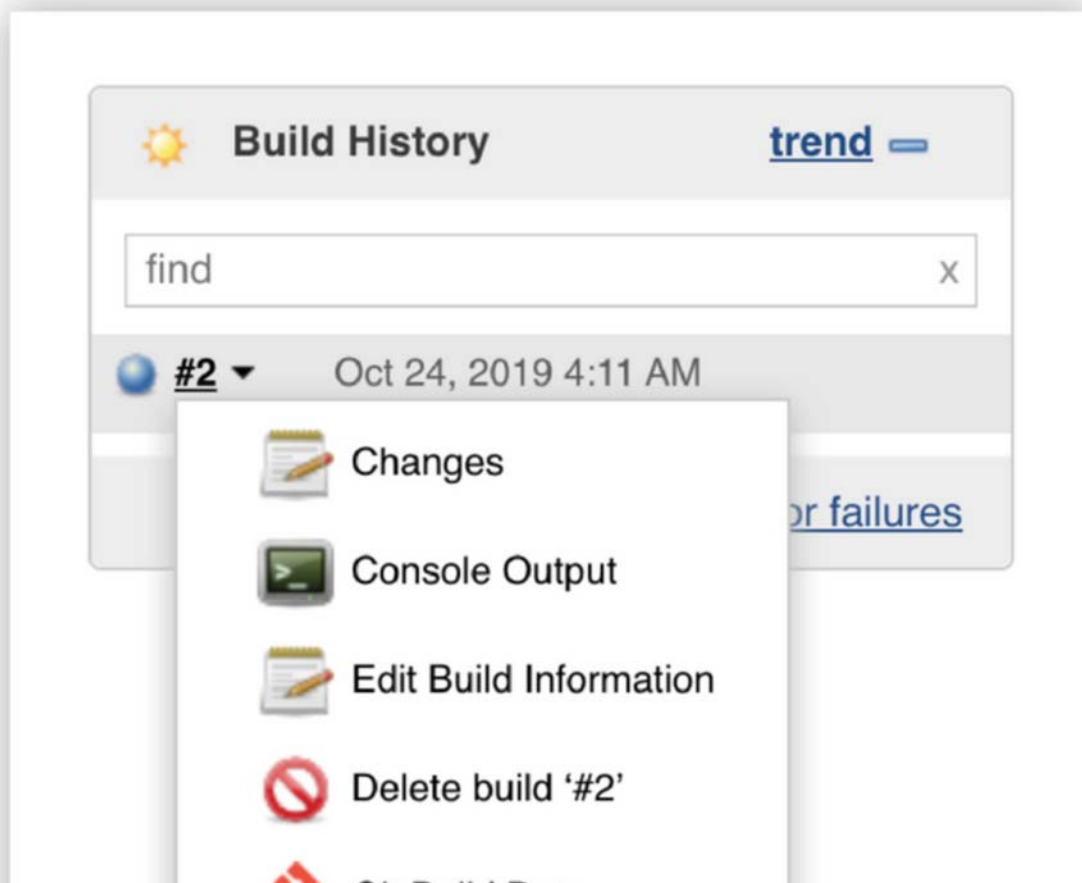
Apply



On the left-hand side, click **Build Now** to start the job.



You can see the job running in the left column. Move your mouse over the build number to get a pulldown menu that includes a link to the Console Output.







Git Build Data



No Tags

The console is where you can see all of the output from the build job. This is a great way to see what is happening if the build shows up with a red circle, indicating that the build has failed.



## Console Output

```
Started by user John Smith
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/BuildAppJob
No credentials specified
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/chasemarellan/deploy-sample.git #
timeout=10
Fetching upstream changes from https://github.com/chasemarellan/deploy-sample.git
> git --version # timeout=10
> git fetch --tags --progress -- https://github.com/chasemarellan/deploy-sample.git
+refs/heads/*:refs/remotes/origin/*
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision e9daf045c7b3d2c8191a1015382e088c6007ba59
(refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f e9daf045c7b3d2c8191a1015382e088c6007ba59
Commit message: "Cleanup"
First time build. Skipping changelog.
[BuildAppJob] $ /bin/sh -xe /tmp/jenkins1972767045004220742.sh
+ ./buildscript.sh
Sending build context to Docker daemon 6.144kB

Step 1/7 : FROM python
----> d6a7b0694364
Step 2/7 : RUN pip install flask
```

Click the Jenkins link and New Item to start a new job, then create another Freestyle job, this time called TestAppJob. This time, leave Source Code Management as None because you have already done all that in the previous job. But you have the option to set a Build Trigger so that this job runs right after the previous job, BuildAppJob.

**Source Code Management**

☒ None

☐ Git

☐ Subversion

**Build Triggers**

☐ Trigger builds remotely (e.g., from scripts)

☒ Build after other projects are built

Projects to watch:

☒ Trigger only if build is stable

☐ Trigger even if the build is unstable

☐ Trigger even if the build fails

☐ Build periodically

☐ GitHub hook trigger for GITScm polling

☐ Poll SCM

Next scroll down and once again add a Build Step of Execute shell script.

Add the following script as the command, using the IP address of an example Jenkins server. (In this case it is set up locally.)

```
if [ "$(curl localhost:8000/test)" = "You are calling me from 172.17.0.1" ]; then
    exit 0
else
    exit 1
fi
```

**Build**

**Execute shell**

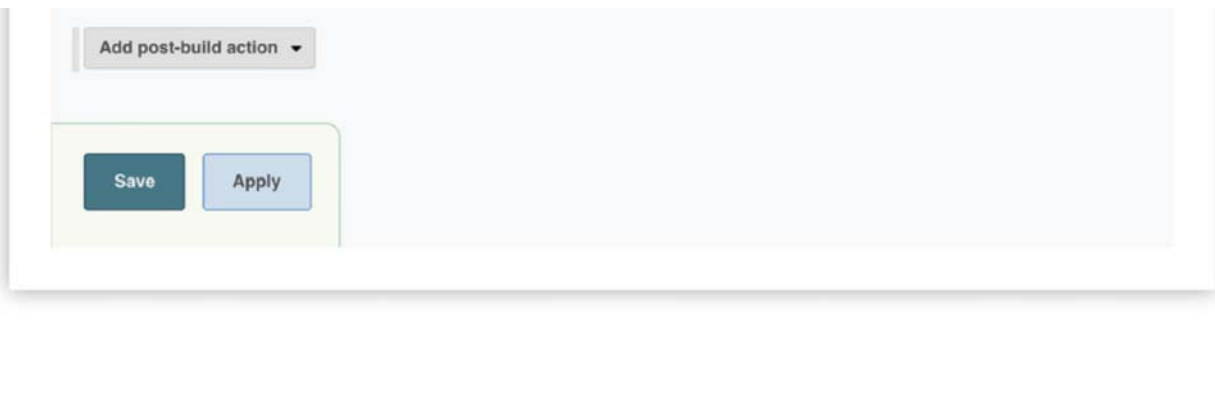
Command: `if [ "$(curl localhost:8000/test)" = "You are calling me from 172.17.0.1" ]; then  
 exit 0  
else  
 exit 1  
fi`

See [the list of available environment variables](#)

Advanced...

Add build step ▾

**Post-build Actions**



Taking this example step by step, first, check to see if a condition is true. That condition is whether the output of calling the test URL gives you back text that says "You are calling me from" followed by your IP address. Remember, the routine is running on the Jenkins server, so that is the IP address that you want in your script.

If that comes back correctly, exit with a code of 0, which means that there are no errors. This means the build script was successful. If it does not come back correctly, exit with a value of 1, which signals an error.

Next, you will look at the console output to see the results.

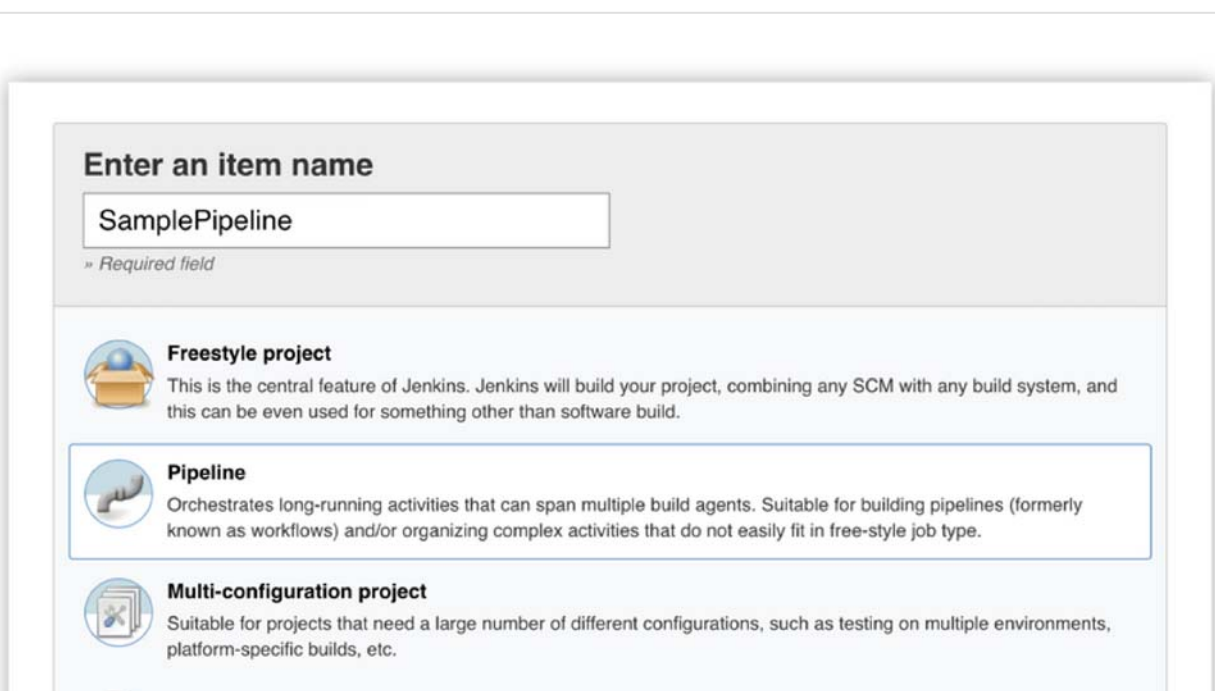
Now, put these jobs together into a pipeline.

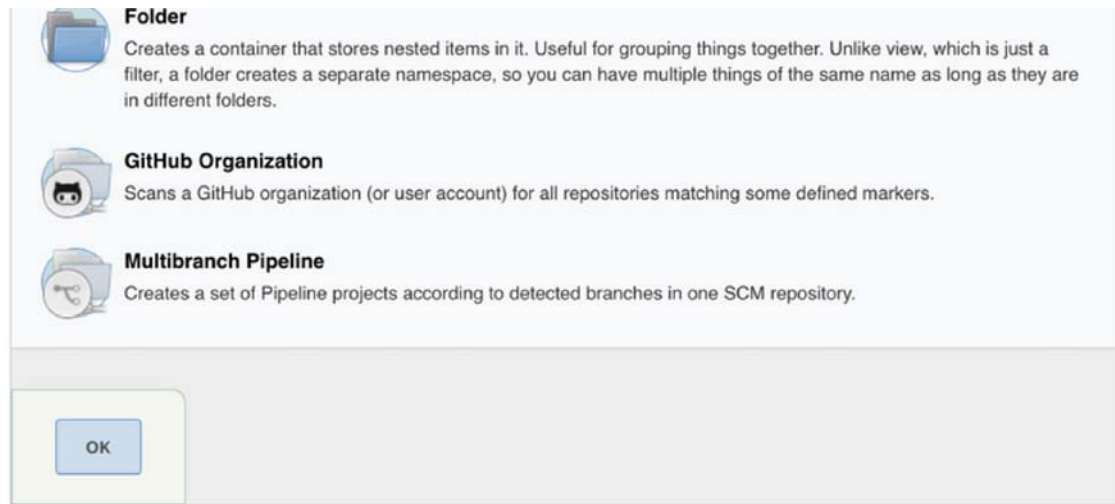
### 6.3.5

## Example Pipeline in Jenkins



Now that you have seen how these two jobs are built, look at how to build an actual pipeline. This examples shows a third New Item, and this time Pipeline is the selected type.





**Folder**  
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**GitHub Organization**  
Scans a GitHub organization (or user account) for all repositories matching some defined markers.

**Multibranch Pipeline**  
Creates a set of Pipeline projects according to detected branches in one SCM repository.

OK

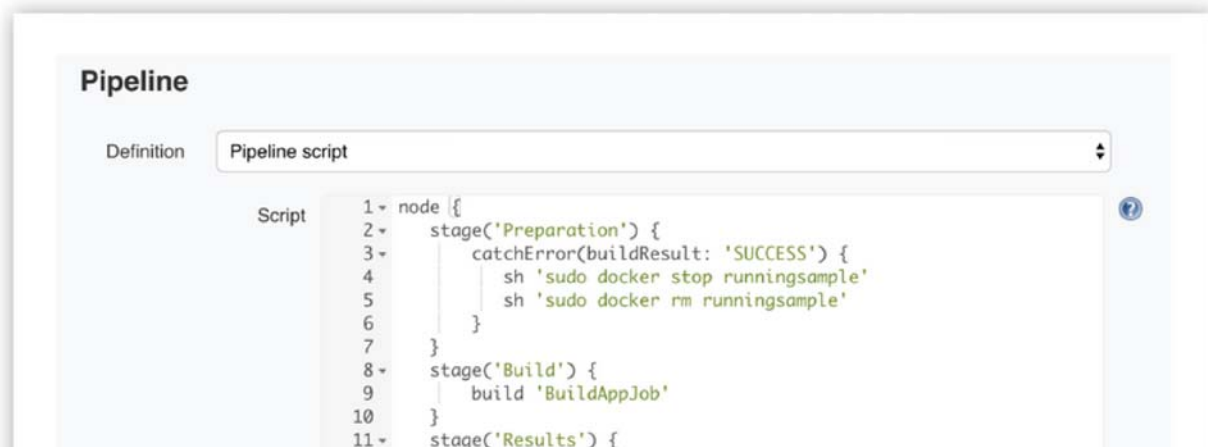
You will notice that you have a number of different ways to trigger a pipeline, but in this case, just trigger it manually.



### Build Triggers

- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ GitHub hook trigger for GITScm polling
- ☐ Poll SCM
- ☐ Disable this project
- ☐ Quiet period
- ☐ Trigger builds remotely (e.g., from scripts)

Look at the pipeline section, scrolling down.



### Pipeline

Definition: Pipeline script

Script

```
1 node {  
2   stage('Preparation') {  
3     catchError(buildResult: 'SUCCESS') {  
4       sh 'sudo docker stop runningsample'  
5       sh 'sudo docker rm runningsample'  
6     }  
7   }  
8   stage('Build') {  
9     build 'BuildAppJob'  
10  }  
11  stage('Results') {
```



This code can go into the script box:

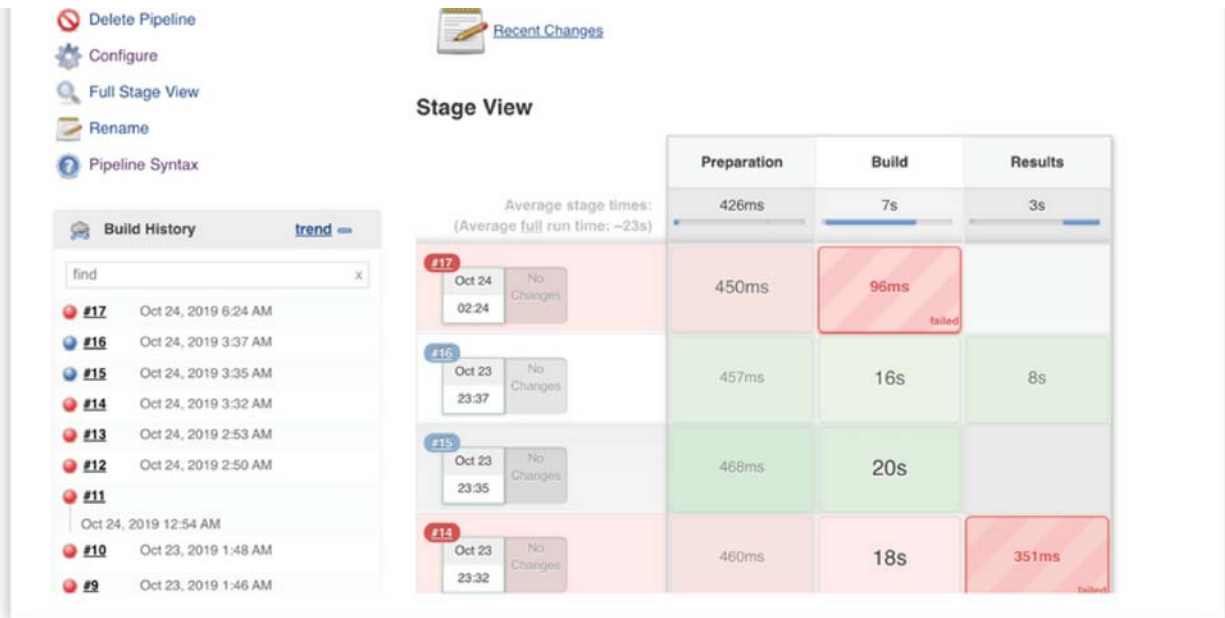
```
node {
  stage('Preparation') {
    catchError(buildResult: 'SUCCESS') {
      sh 'sudo docker stop runningsample'
      sh 'sudo docker rm runningsample'
    }
  }
  stage('Build') {
    build 'BuildAppJob'
  }
  stage('Results') {
    build 'TestAppJob'
  }
}
```

Look at this one step at a time. First, you are executing this pipeline on a single node. The pipeline itself has three stages, Preparation, Build, and Results. These stages that are run sequentially. If an early stage fails, the pipeline stops.

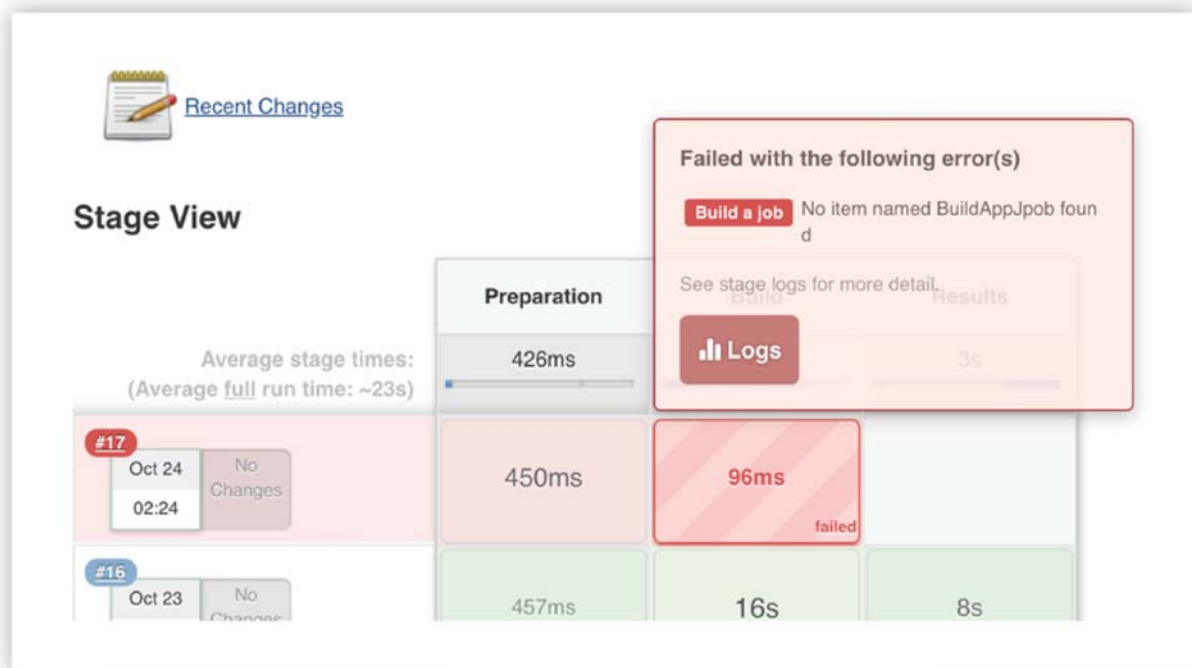
Use the Preparation stage to stop and remove the container if it is already running. If there is not yet a running container, you will get an error, so set the script to catch any errors and return a “SUCCESS” value. This way, the pipeline continues on.

The next stage (the Build stage) you can simply call the BuildAppJob. If it succeeds, you will move on to calling the TestAppJob. The results of TestAppJob will determine whether the pipeline itself succeeds or fails. Click save and Build Now to run the pipeline.





Each time you build the pipeline, you will see the success or failure of each stage. If you do have a failure, you can easily see what is happening by running your mouse over the offending stage:



To make changes, click the Configure link, fix any problems, and save and build again.





Now, all of those build jobs and pipeline pieces have come together as CI/CD components in a line to get a set of jobs or scripts done. While it is a simple example, it can come in handy when you want to inspect other's build jobs or start building your own CI/CD pipeline.

6.3.6

## Lab – Build a CI/CD Pipeline Using Jenkins



In this lab, you will commit the Sample App code to a GitHub repository, modify the code locally, and then commit your changes. You will then install a Docker container that includes the latest version of Jenkins. You will configure Jenkins and then use Jenkins to download and run your Sample App program. Next, you will create a testing job inside Jenkins that will verify your Sample App program successfully runs each time you build it. Finally, you will integrate your Sample App and testing job into a Continuous Integration/Continuous Development pipeline that will verify your Sample App is ready to be deployed each time you change the code.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Commit the Sample App to Git
- Part 3: Modify the Sample App and Push Changes to Git
- Part 4: Download and Run the Jenkins Docker Image
- Part 5: Configure Jenkins
- Part 6: Use Jenkins to Run a Build of Your App
- Part 7: Use Jenkins to Test a Build
- Part 8: Create a Pipeline in Jenkins



Build a CI-CD Pipeline Using Jenkins



6.3.7

## Project Activity 4: Automated Software Testing and Deployment



In this activity, you will complete the following tasks:

- Design an automated testing process.
- Implement your automated testing process.

Refer to the **DEVASC Project Rubric** below for this activity to record your process and outcomes.

### Scenario

The marketing department has received very favorable reviews and feedback about your application. Your manager has acquired funding and resources for your team to add some of the backlog features to your application.

As your team continues working on the code, each team member contributes to your shared application codebase on GitHub with their changes. Despite the fact that you are trying to do doing manual code reviews for each commit, or pull request, sometimes a team member's change will break some functionality in the application. Many times, such issues are identified only when the application is deployed and running. This usually leads to long troubleshooting sessions and slows down the development.

To eliminate the issues from above and ease development, your team has been asked to use some of the best practices from the DevOps methodology. Design a testing process that would be triggered automatically by GitHub whenever there is a new commit to the codebase. Only if all the test cases are passed, can the committed change be merged, or further used. Your manager has recommended that, in addition to Jenkins, explore other CI/CD tools like GitHub Actions, CircleCI, etc., that might help you to build this automated CI/CD pipeline.

#### **Deliverable/Rubric:**

- Which features did your team choose to implement from your backlog?
- What are the specific objectives of these features?
- Why were these features chosen?
- Document your team member roles, knowledge and skillsets, if anything has changed in relation to this new Project Activity.
- Provide a brief description of your team strategy for completing this project if anything has changed in relation to this new Project Activity.

### Final Deliverables

At this point, your project has been completed. Your team will present to your manager:

- Presentation
- Team activities and reflection

## Presentation

**Deliverable/Rubric:** Create a presentation about the project you selected. Your presentation should include:

- Application code with new features
- Show changes pushed to GitHub
- Provide test cases used
- Describe CI/CD pipeline functionality
- Reflection points – what issues have you faced while working on this activity, how did you find solutions, what have you learned, etc.

## Team Activities and Reflection

**Deliverable/Rubric:** Your manager is interested in knowing how everyone is continuing to work together as a team. Here is a list of questions from your manager:

- What did you enjoy about working as a team? What worked well?
- What team problems did you encounter and how did you resolve them?
- What technical problems did you encounter and how did you resolve them?
- How was each team member held accountable individually and for the team as a whole?
- What was your team's decision-making process?
- Overall, how were the team dynamics and what were any lessons learned?

 Project Activity 4 – Automated Software Testing an...



6.2

[Creating and Deploying a Sample Applicat...](#)[Networks for Application Development an...](#)

6.4

