Jenkins on Azure documentation

Learn how to use Jenkins to automate continuous integration and continuous delivery on Azure.

About Jenkins on Azure

OVERVIEW

About Jenkins on Azure

REFERENCE

Jenkins plug-ins for Azure

ARCHITECTURE

Jenkins architecture

Install and configure Jenkins

QUICKSTART

Configure Jenkins using Azure CLI

DOWNLOAD

Sample jobs and scripts

Scale using Jenkins and Azure

TUTORIAL

Scale with Azure Container Instances

Scale with Azure VM agents

CI/CD to Azure Functions

TUTORIAL

Deploy to Azure Functions

CI/CD to AKS

TUTORIAL

Deploy from GitHub to AKS

CI/CD to App Service

TUTORIAL

Create Azure resources in a pipeline job

CI/CD to Azure DevOps

TUTORIAL

Deploy to a Linux VM using Azure DevOps Services

CI/CD to Azure Storage

TUTORIAL

Using Azure Storage with a Jenkins C/I solution

CI/CD to Linux VMs

TUTORIAL

Create a dev infrastructure on a Linux VM

CI/CD to Service Fabric

TUTORIAL

Build and deploy an Azure Service Fabric app

CI/CD to Azure Spring Apps

TUTORIAL

Deploy to Azure Spring Apps using the Azure CLI

Azure and Jenkins

Article • 02/11/2021

Jenkins is a popular open-source automation server used to set up continuous integration and delivery (CI/CD) for your software projects. You can host your Jenkins deployment in Azure or extend your existing Jenkins configuration using Azure resources. Jenkins plug-ins are also available to simplify CI/CD of your applications to Azure.

This article is an introduction to using Azure with Jenkins, detailing the core Azure features available to Jenkins users. For more information about getting started with your own Jenkins server in Azure, see Create a Jenkins server on Azure.

Host your Jenkins servers in Azure

Host Jenkins in Azure to centralize your build automation and scale your deployment as the needs of your software projects grow. See Quickstart - Get started with Jenkins to learn how to install and configure Jenkins on a Linux VM. Monitor and manage your Azure Jenkins deployment using Azure Monitor logs and the Azure CLI.

Scale your build automation on demand

Add build agents to your existing Jenkins deployment to scale your Jenkins build capacity as the number of builds and complexity of your jobs and pipelines increase. You can run these build agents on Azure virtual machines by using the Azure VM Agents plug-in. See our tutorial for more details.

Once configured with an Azure service principal, Jenkins jobs and pipelines can use this credential to:

- Securely store and archive build artifacts in Azure Storage using the Azure Storage plug-in. Review the Jenkins storage how-to to learn more.
- Manage and configure Azure resources with the Azure CLI.

Deploy your code into Azure services

Use Jenkins plug-ins to deploy your applications to Azure as part of your Jenkins CI/CD pipelines. Deploying into Azure App Service and Azure Container Service lets you stage, test, and release updates to your applications without managing the underlying infrastructure.

Plug-ins are available to deploy to the following services and environments:

- Azure App Service on Linux. See the tutorial to get started.

Get Started: Install Jenkins on an Azure Linux VM

Article • 05/30/2023

This article shows how to install Jenkins on an Ubuntu Linux VM with the tools and plug-ins configured to work with Azure.

In this article, you'll learn how to:

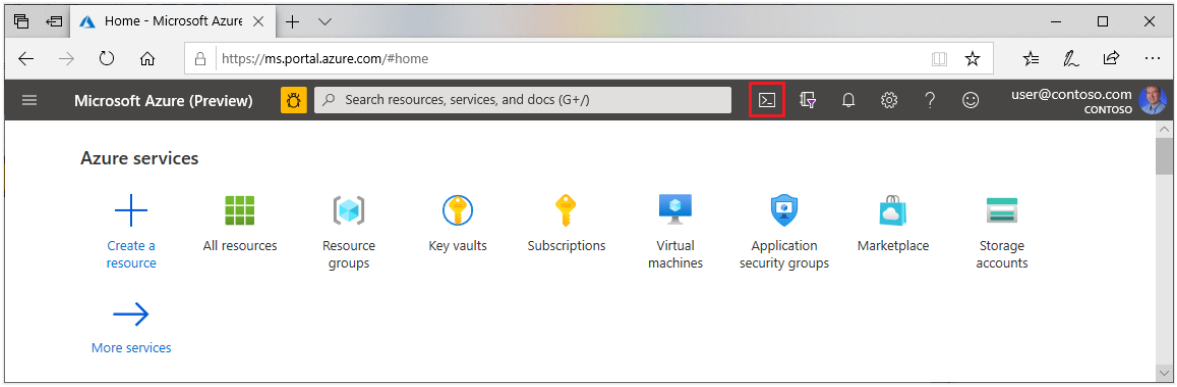
> - Create a setup file that downloads and installs Jenkins
> - Create a resource group
> - Create a virtual machine with the setup file
> - Open port 8080 in order to access Jenkins on the virtual machine
> - Connect to the virtual machine via SSH
> - Configure a sample Jenkins job based on a sample Java app in GitHub
> - Build the sample Jenkins job

1. Configure your environment

- **Azure subscription**: If you don't have an Azure subscription, create a free account before you begin.

2. Open Cloud Shell

1. If you already have a Cloud Shell session open, you can skip to the next section.
2. Browse to the Azure portal
3. If necessary, log in to your Azure subscription and change the Azure directory.
4. Open Cloud Shell.



5. If you haven't previously used Cloud Shell, configure the environment and storage settings.

6. Select the command-line environment.

3. Create a virtual machine

1. Create a test directory called `jenkins-get-started`.

2. Switch to the test directory.

3. Create a file named `cloud-init-jenkins.txt`.

4. Paste the following code into the new file:

JSON

```
#cloud-config
package_upgrade: true
runcmd:
  - sudo apt install openjdk-11-jre -y
  - curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
| sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null
  -  echo 'deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]
https://pkg.jenkins.io/debian-stable binary/' | sudo tee
/etc/apt/sources.list.d/jenkins.list > /dev/null
  - sudo apt-get update && sudo apt-get install jenkins -y
  - sudo service jenkins restart
```

5. Run az group create to create a resource group.

Azure CLI

```
az group create --name jenkins-get-started-rg --location eastus
```

6. Run az vm create to create a virtual machine.

Azure CLI

```
az vm create \
--resource-group jenkins-get-started-rg \
--name jenkins-get-started-vm \
--image UbuntuLTS \
--admin-username "azureuser" \
--generate-ssh-keys \
--public-ip-sku Standard \
--custom-data cloud-init-jenkins.txt
```

7. Run az vm list to verify the creation (and state) of the new virtual machine.

Azure CLI

```
az vm list -d -o table --query "[?name=='jenkins-get-started-vm']"
```

8. As Jenkins runs on port 8080, run az vm open to open port 8080 on the new virtual machine.

Azure CLI

```
az vm open-port \
--resource-group jenkins-get-started-rg \
--name jenkins-get-started-vm  \
--port 8080 --priority 1010
```

4. Configure Jenkins

1. Run az vm show to get the public IP address for the sample virtual machine.

Azure CLI

```
az vm show \
--resource-group jenkins-get-started-rg \
--name jenkins-get-started-vm -d \
--query [publicIps] \
--output tsv
```

Key points:

- The `--query` parameter limits the output to the public IP addresses for the virtual machine.

2. Using the IP address retrieved in the previous step, SSH into the virtual machine. You'll need to confirm the connection request.

   Azure CLI

   ```
   ssh azureuser@<ip_address>
   ```

   Key points:

   - Upon successful connection, the Cloud Shell prompt includes the user name and virtual machine name: `azureuser@jenkins-get-started-vm`.

3. Verify that Jenkins is running by getting the status of the Jenkins service.

   Bash

   ```
   service jenkins status
   ```

   Key points:

   - If you receive an error regarding the service not existing, you may have to wait a couple of minutes for everything to install and initialize.

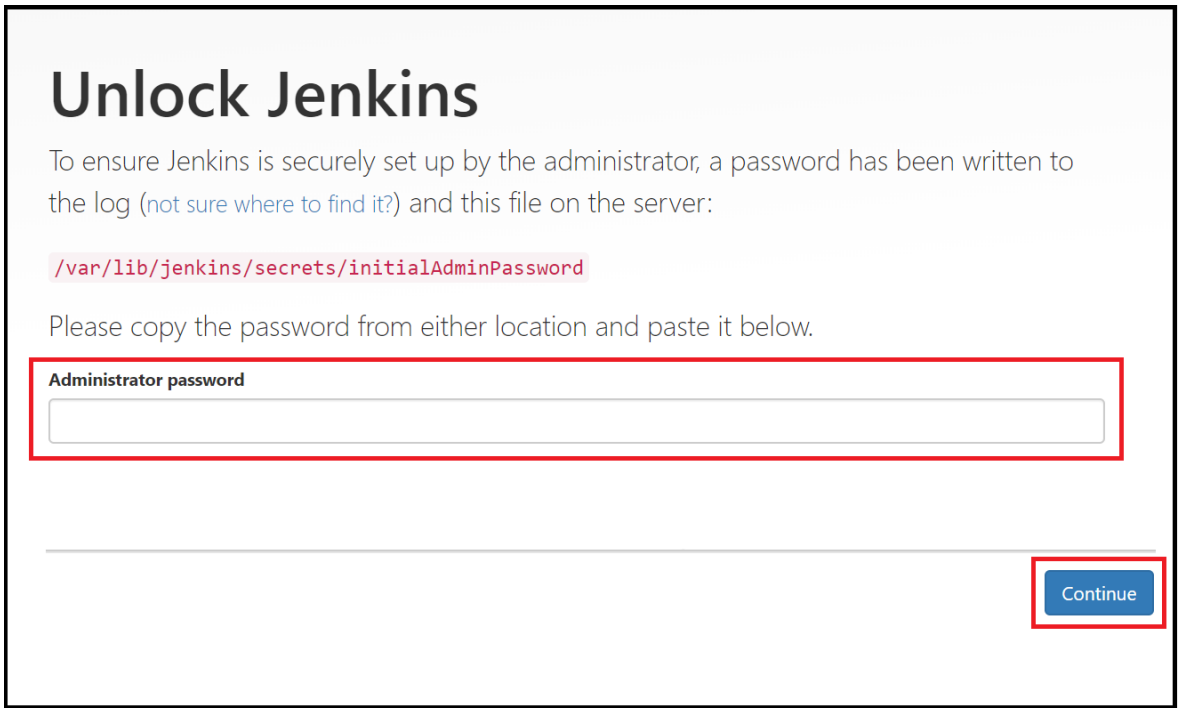
4. Get the autogenerated Jenkins password.

   Bash

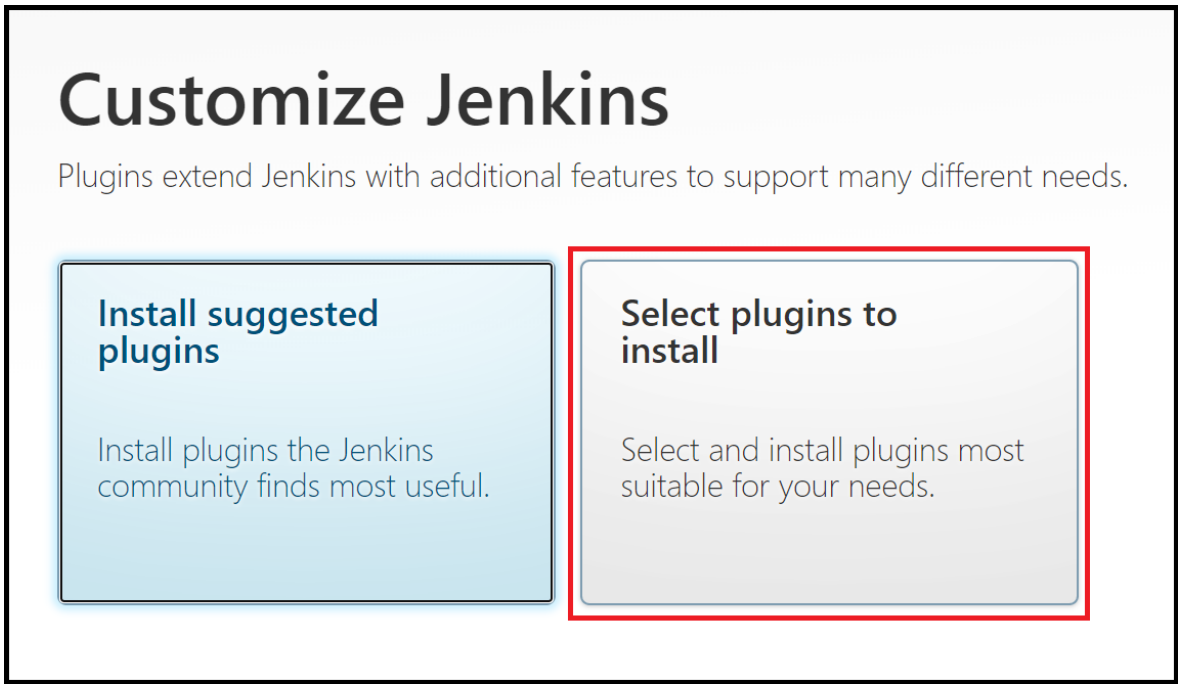
   ```
   sudo cat /var/lib/jenkins/secrets/initialAdminPassword
   ```

5. Using the IP address, open the following URL in a browser: `http://<ip_address>:8080`

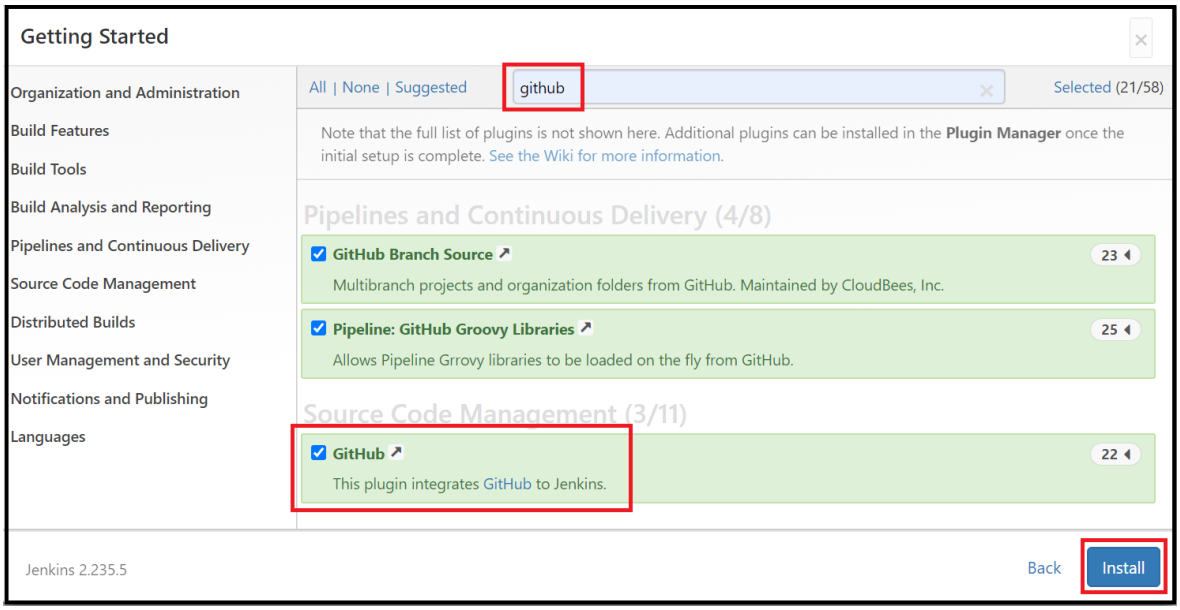
6. Enter the password you retrieved earlier and select **Continue**.



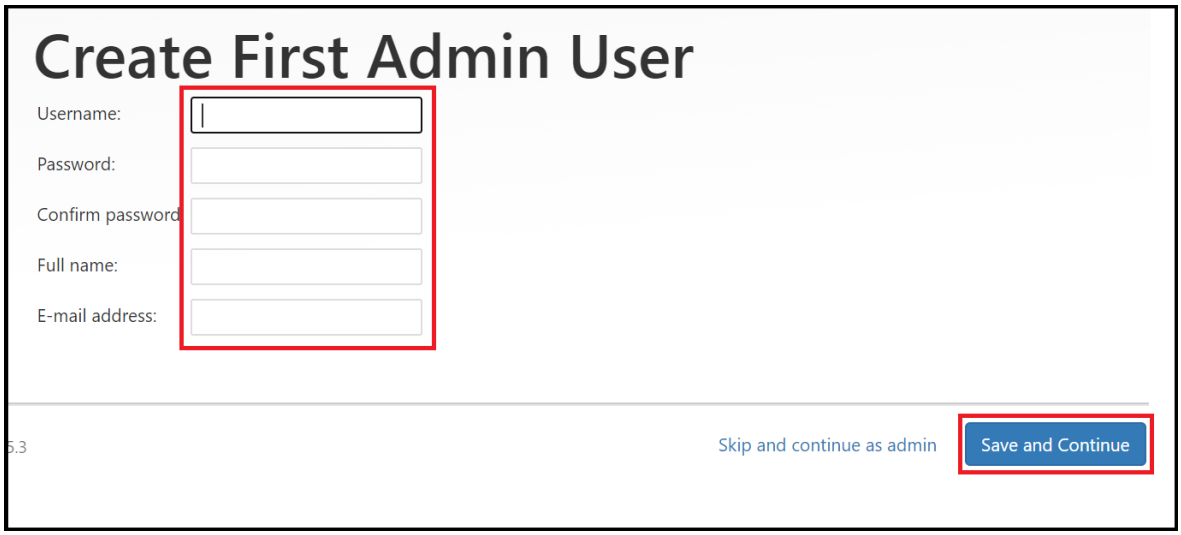
7. Select **Select plug-in to install**.



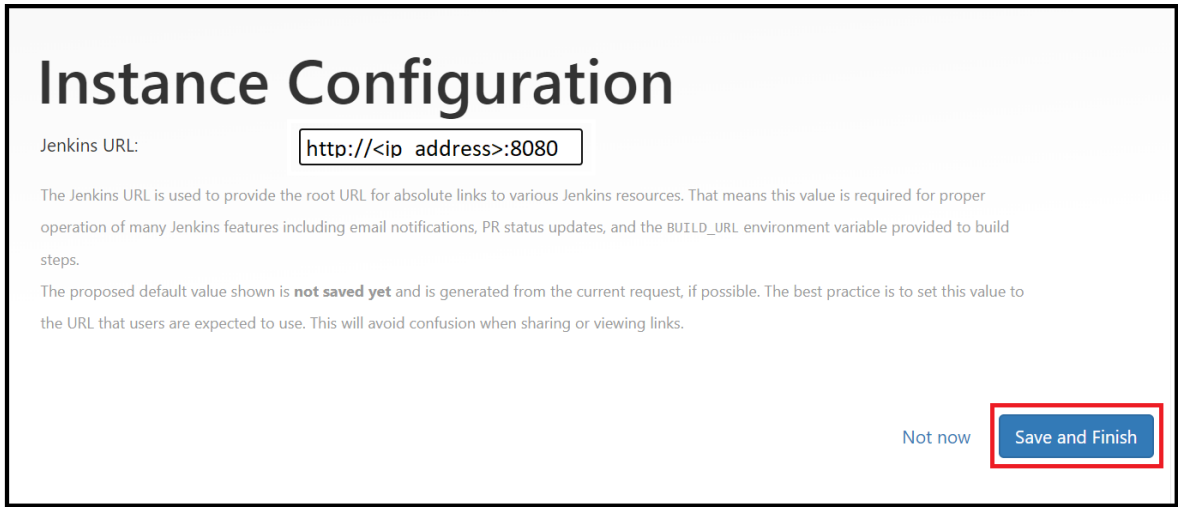
8. In the filter box at the top of the page, enter `github`. Select the GitHub plug-in and select **Install**.



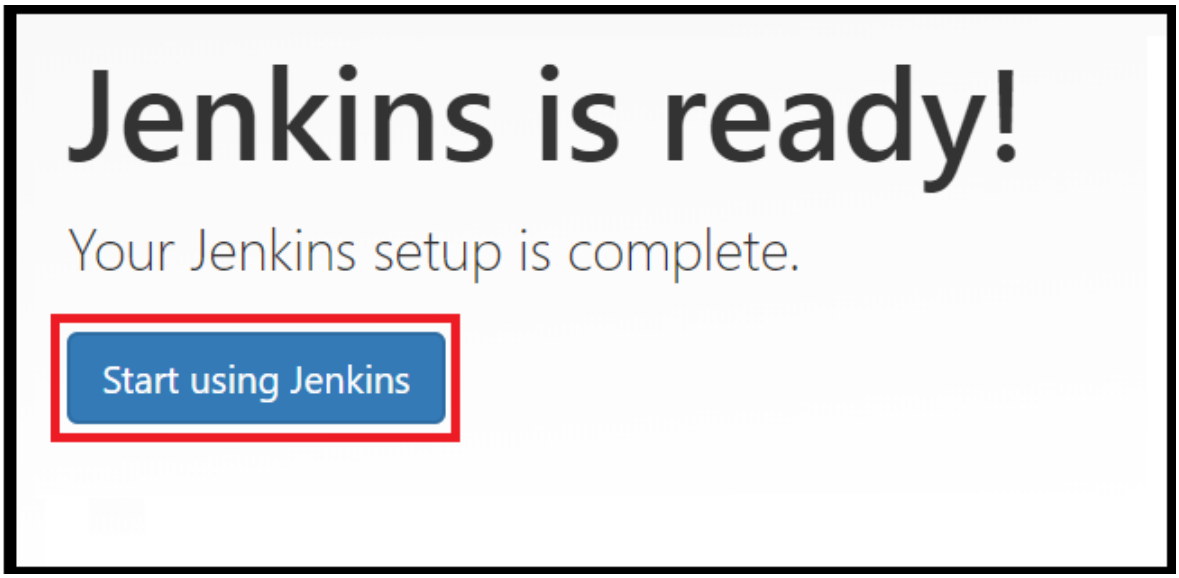
9. Enter the information for the first admin user and select **Save and Continue**.



10. On the **Instance Configuration** page, select **Save and Finish**.

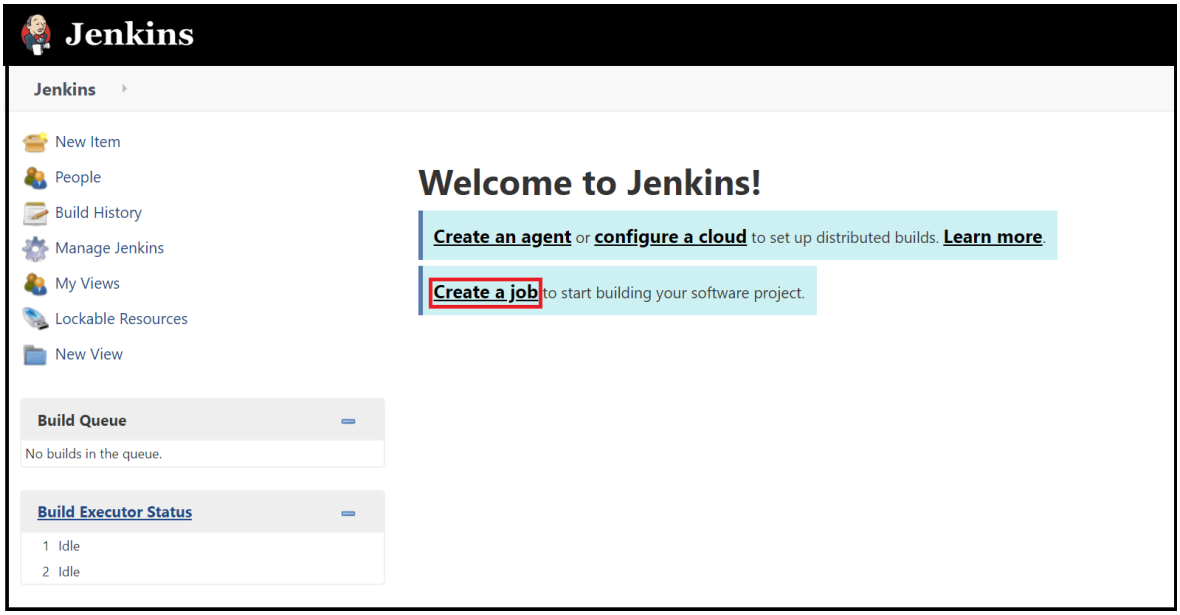


11. Select **Start using Jenkins**.

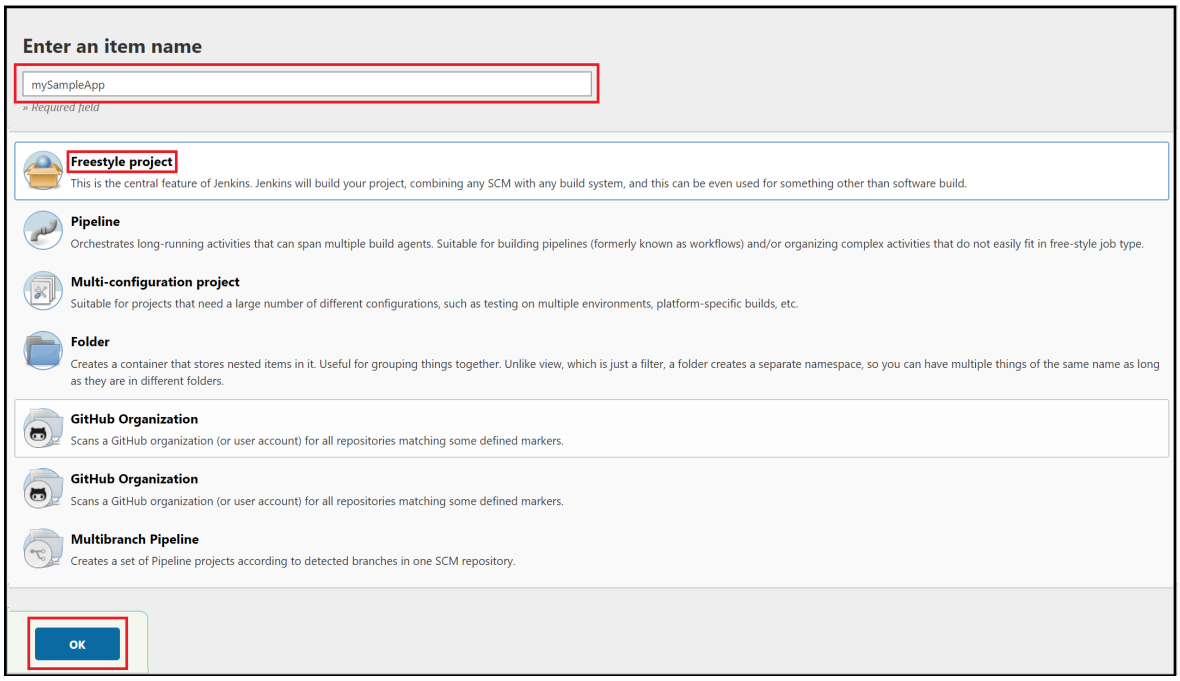


5. Create your first job

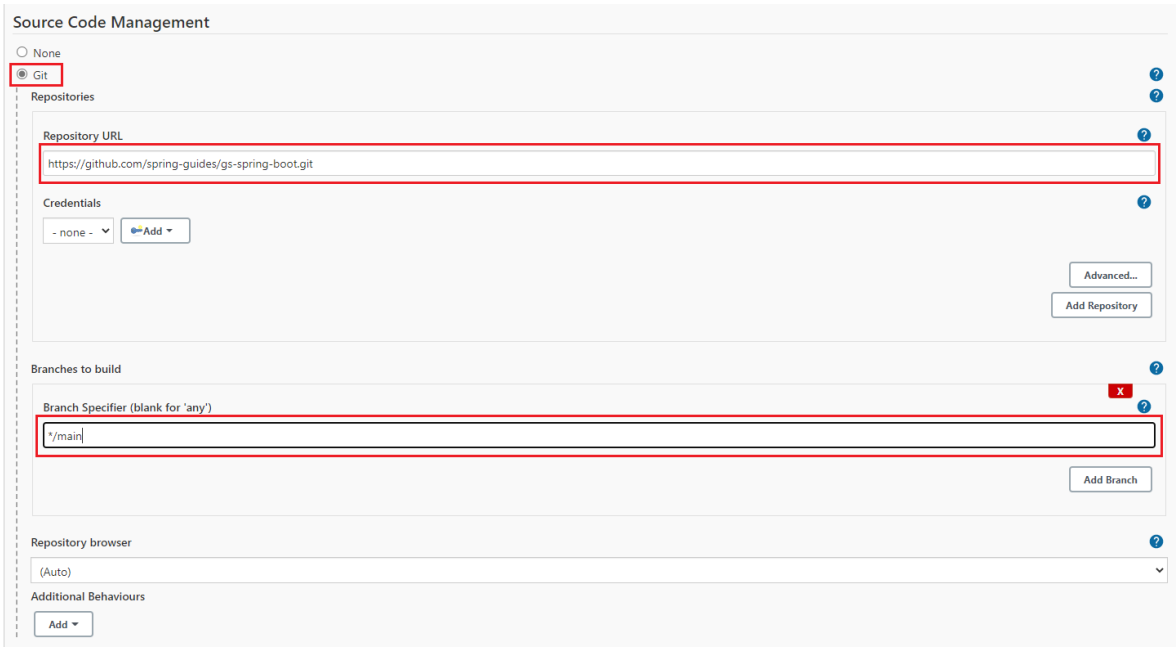
1. On the Jenkins home page, select **Create a job**.



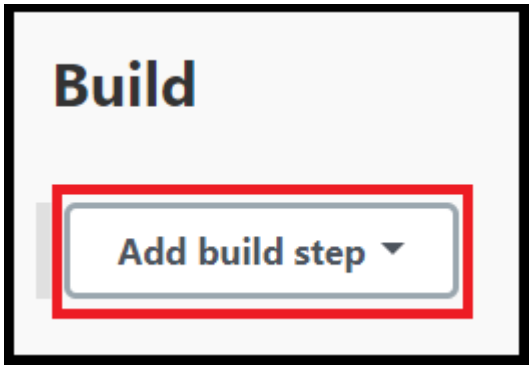
2. Enter a job name of `mySampleApp`, select **Freestyle project**, and select **OK**.



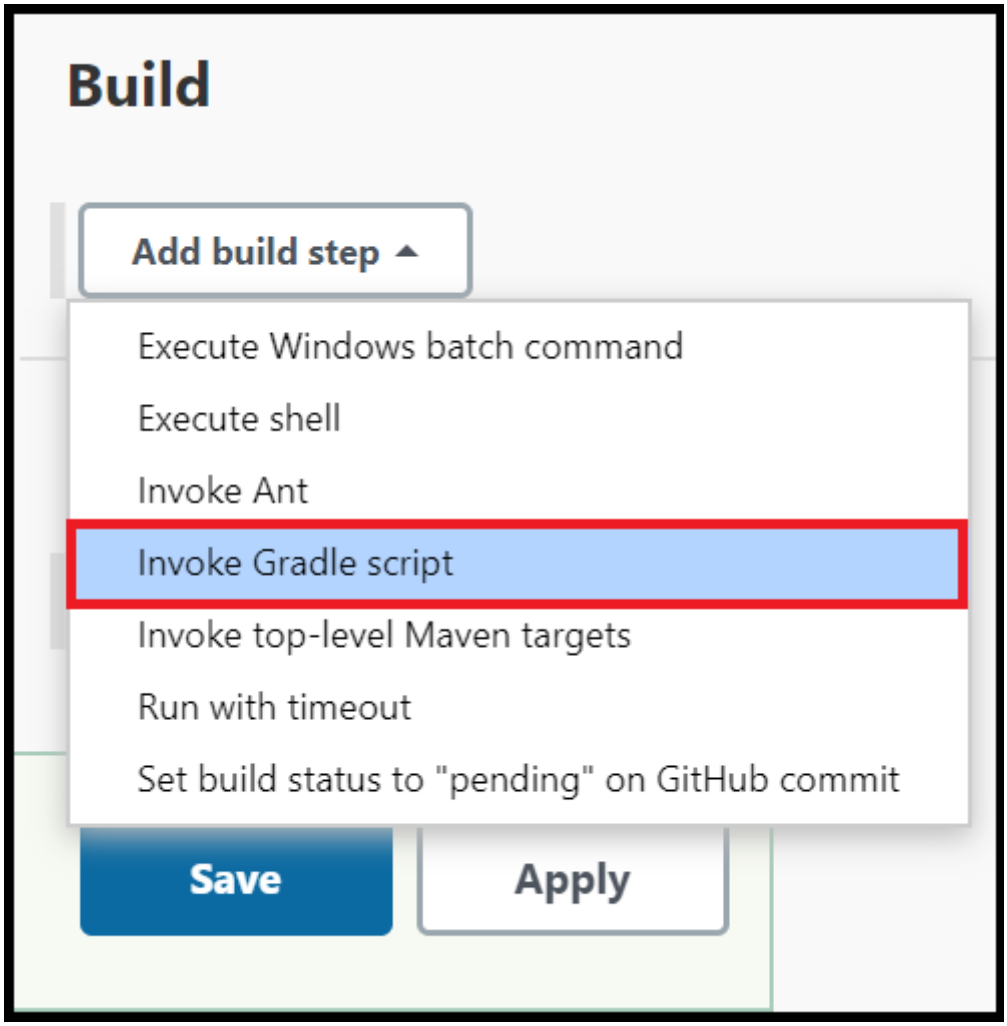
3. Select the **Source Code Management** tab. Enable **Git** and enter the following URL for the **Repository URL** value: `https://github.com/spring-guides/gs-spring-boot.git`. Then change the **Branch Specifier** to `*/main`.



4. Select the **Build** tab, then select **Add build step**

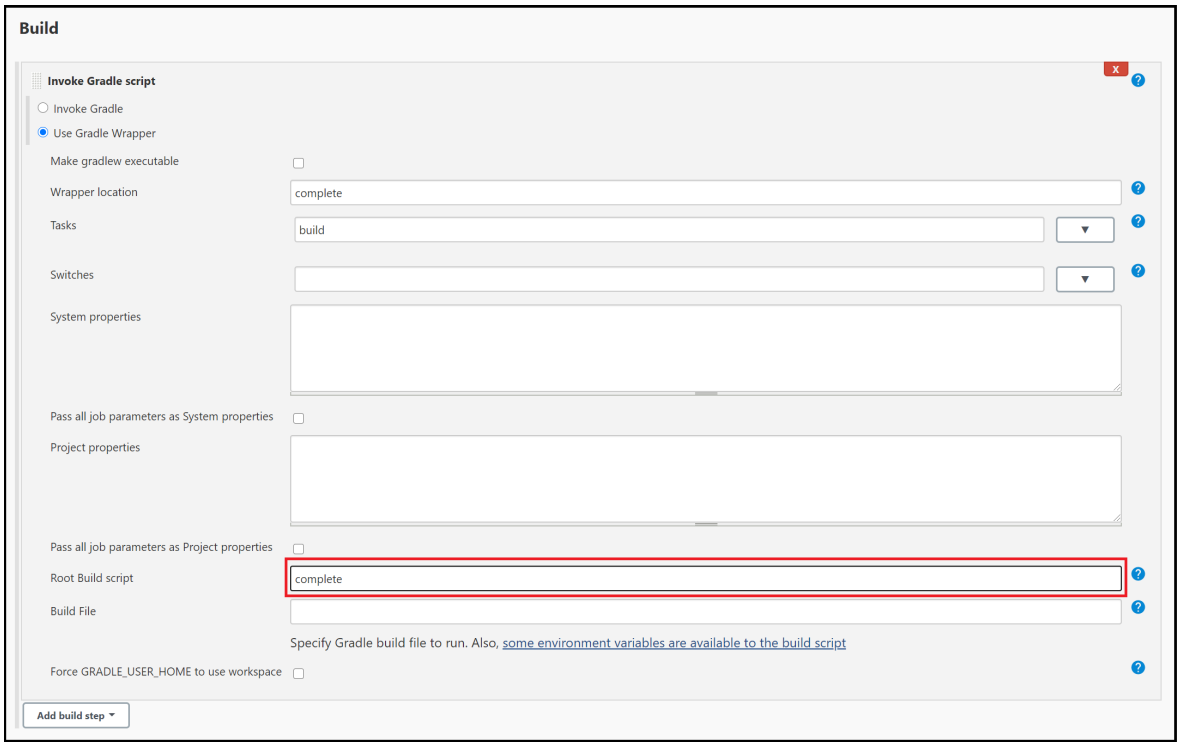


5. From the drop-down menu, select **Invoke Gradle script**.



6. Select **Use Gradle Wrapper**, then enter `complete` in **Wrapper location** and `build` for **Tasks**.

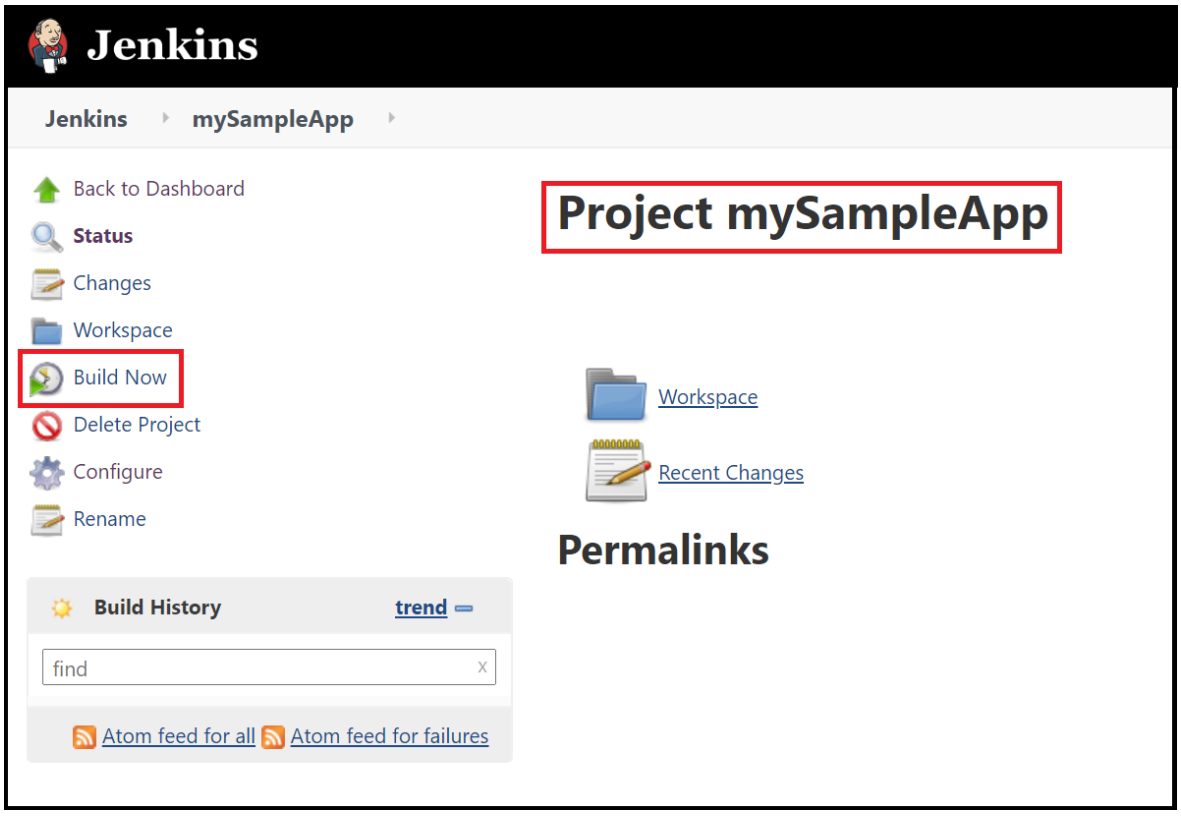
7. Select **Advanced** and enter `complete` in the **Root Build script** field.



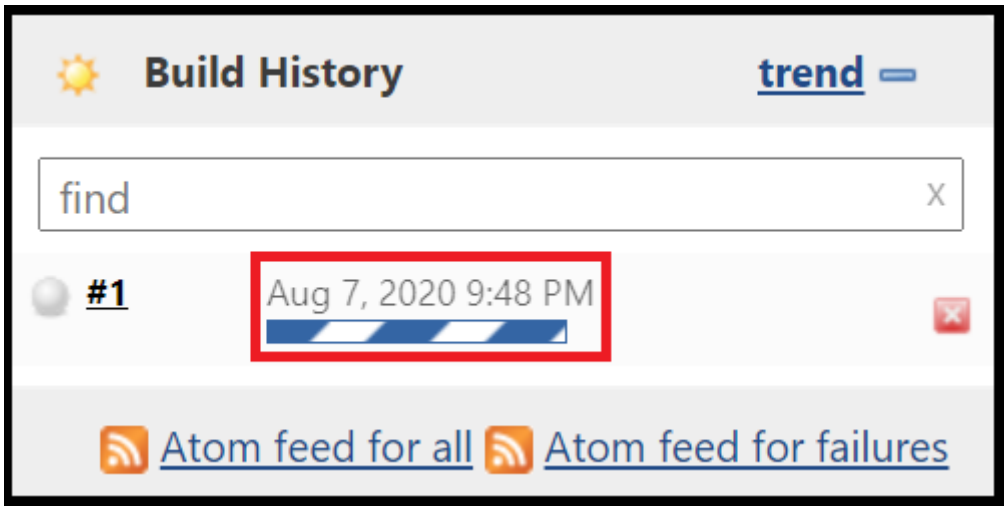
8. Scroll to the bottom of the page, and select **Save**.

6. Build the sample Java app

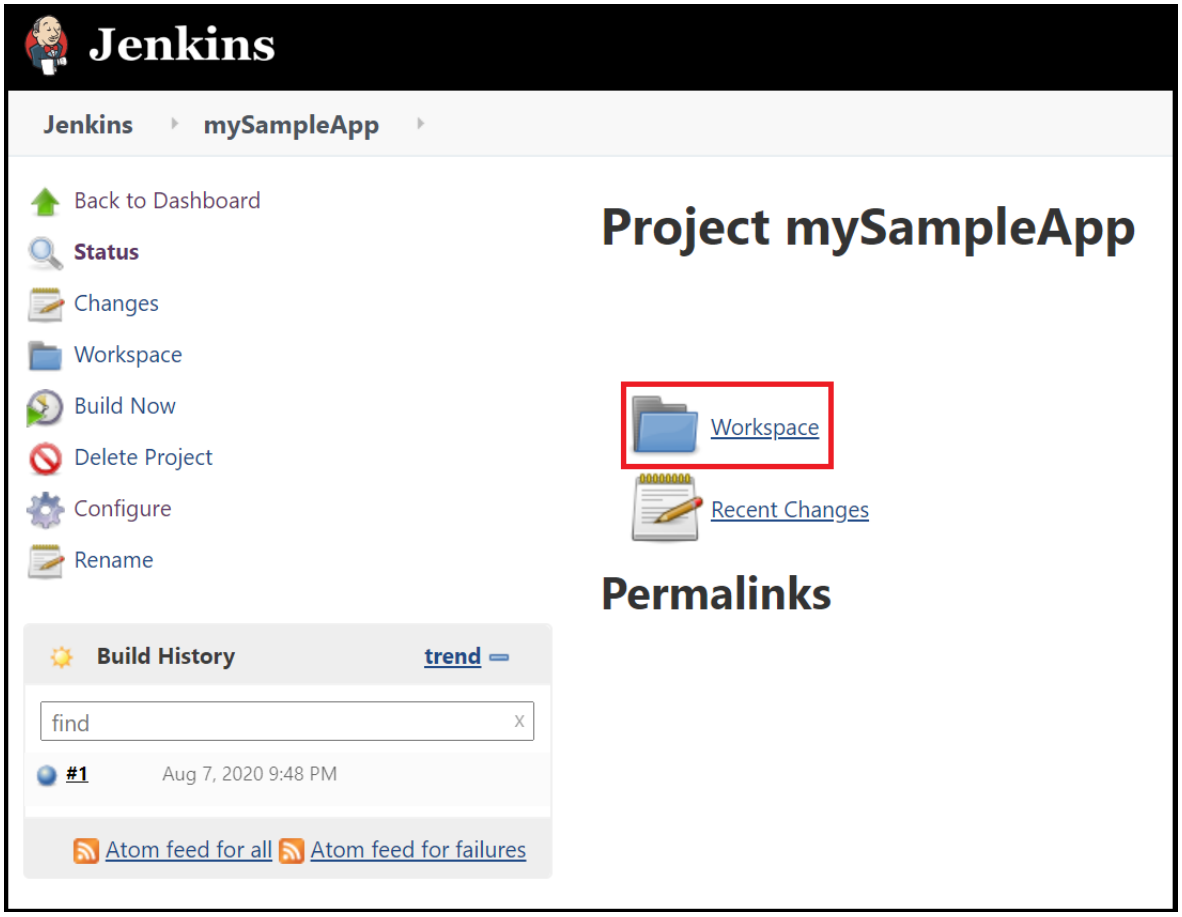
1. When the home page for your project displays, select **Build Now** to compile the code and package the sample app.



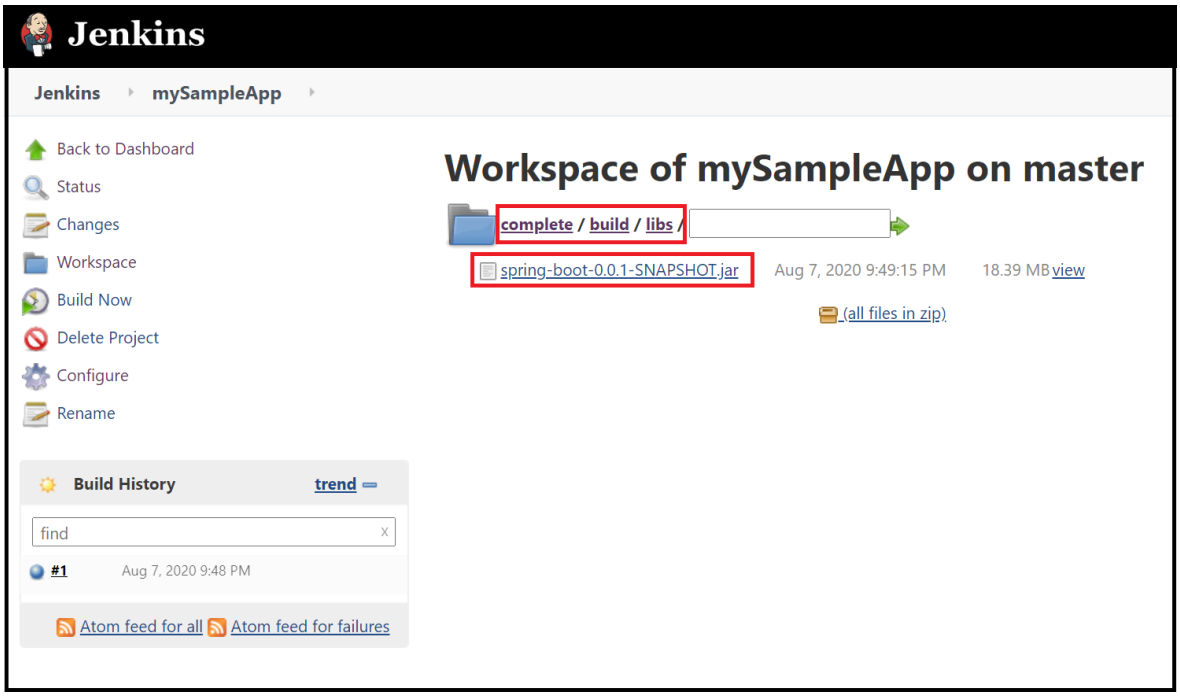
2. A graphic below the **Build History** heading indicates that the job is being built.



3. When the build completes, select the **Workspace** link.



4. Navigate to `complete/build/libs` to see that the `.jar` file was successfully built.



5. Your Jenkins server is now ready to build your own projects in Azure!

Troubleshooting

If you encounter any problems configuring Jenkins, refer to the Jenkins installation page for the latest instructions and known issues.

Next steps

Jenkins on Azure

Tutorial: Use Azure Container Instances as a Jenkins build agent

Article • 08/10/2023

> **ⓘ Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

Azure Container Instances (ACI) provides an on-demand, burstable, and isolated environment for running containerized workloads. Because of these attributes, ACI makes a great platform for running Jenkins build jobs at a large scale. This article shows you how to deploy an ACI and add it as a permanent build agent for a Jenkins controller.

For more information on Azure Container Instances, see About Azure Container Instances.

Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free Azure account before you begin.
- **Jenkins server**: If you don't have a Jenkins server installed, create a Jenkins server on Azure.

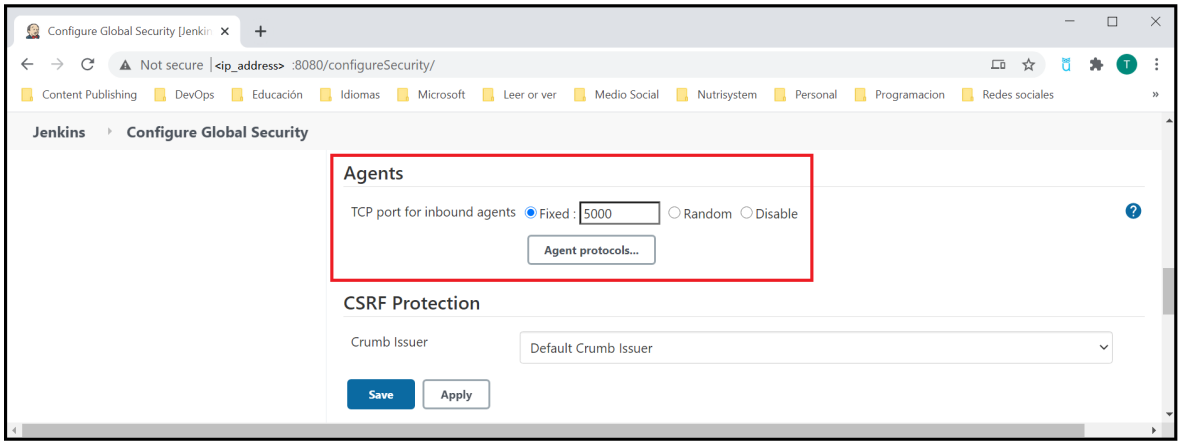
Prepare the Jenkins controller

1. Browse to your Jenkins portal.
2. From the menu, select **Manage Jenkins**.
3. Under **System Configuration**, select **Configure System**.
4. Verify that the **Jenkins URL** is set to the HTTP address of your Jenkins installation - `http://<your_host>.<your_domain>:8080/`.
5. From the menu, select **Manage Jenkins**.

6. Under **Security**, select **Configure Global Security**.

7. Under **Agents**, specify **Fixed** port and enter the appropriate port number for your environment.

   Configuration example:



8. Select **Save**.

Create Jenkins work agent

1. Browse to your Jenkins portal.

2. From the menu, select **Manage Jenkins**.

3. Under **System Configuration**, select **Manage Nodes and Clouds**.

4. From the menu, select **New Node**.

5. Enter a value for **Node Name**.

6. Select **Permanent Agent**.

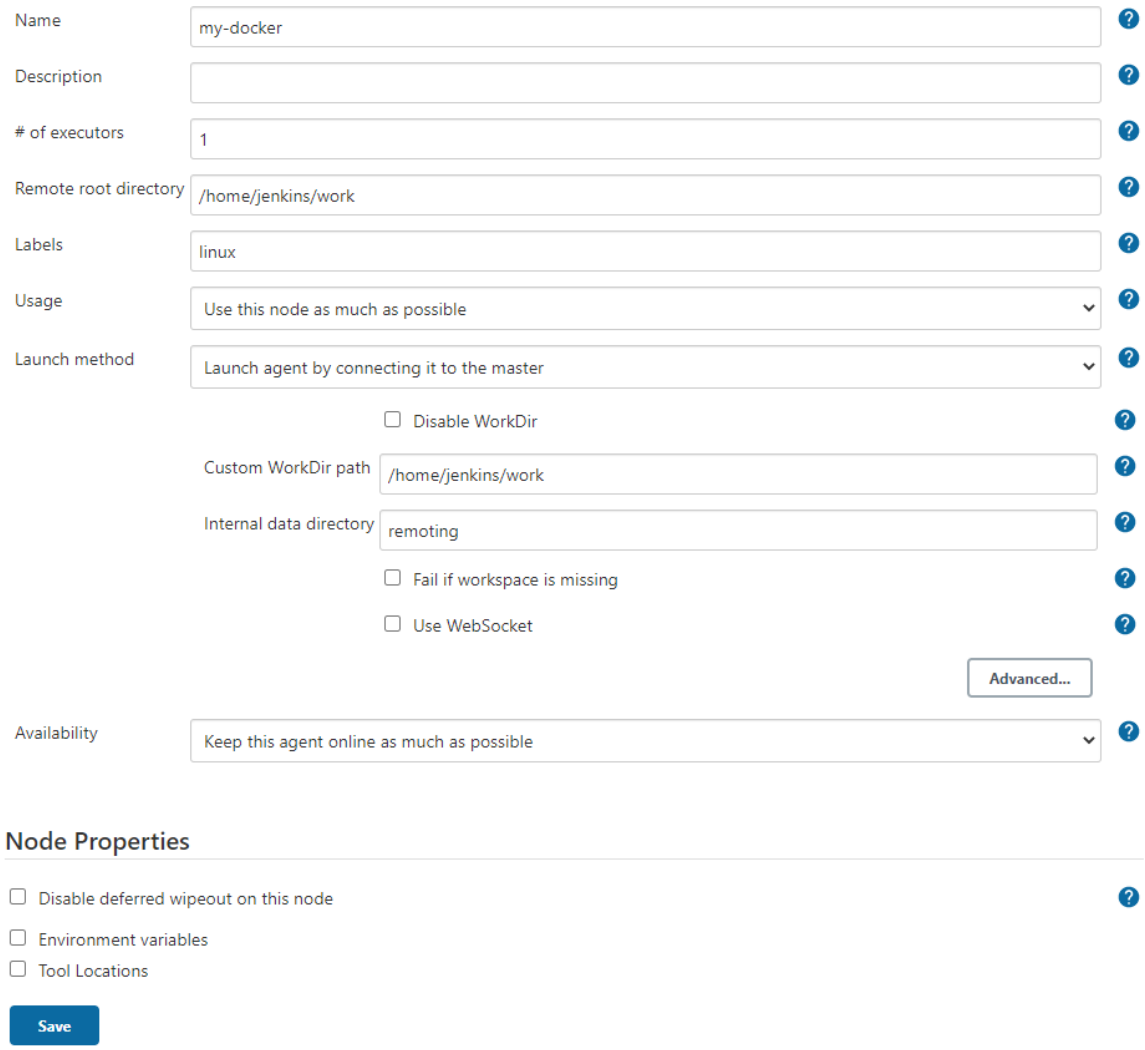
7. Select **OK**.

8. Enter a value for **Remote root directory**. For example, `/home/jenkins/work`

9. Add a **Label** (Labels are used to group multiple agents into one logical group. An example of a label would be `linux` to group your Linux agents.) with the value of `linux`.

10. Set **Launch method** to **Launch agent by connecting to the master**.

11. Verify that all required fields have been specified or entered:



12. Select **Save**.

13. On the agent status page, you should see the `JENKINS_SECRET` and `AGENT_NAME`. The following screen shot shows how to identify the values. Both values are needed when you create the Azure Container Instance.

Create Azure Container Instance with CLI

1. Use az group create to create an Azure resource group.

Azure CLI

```
az group create --name my-resourcegroup --location westus
```

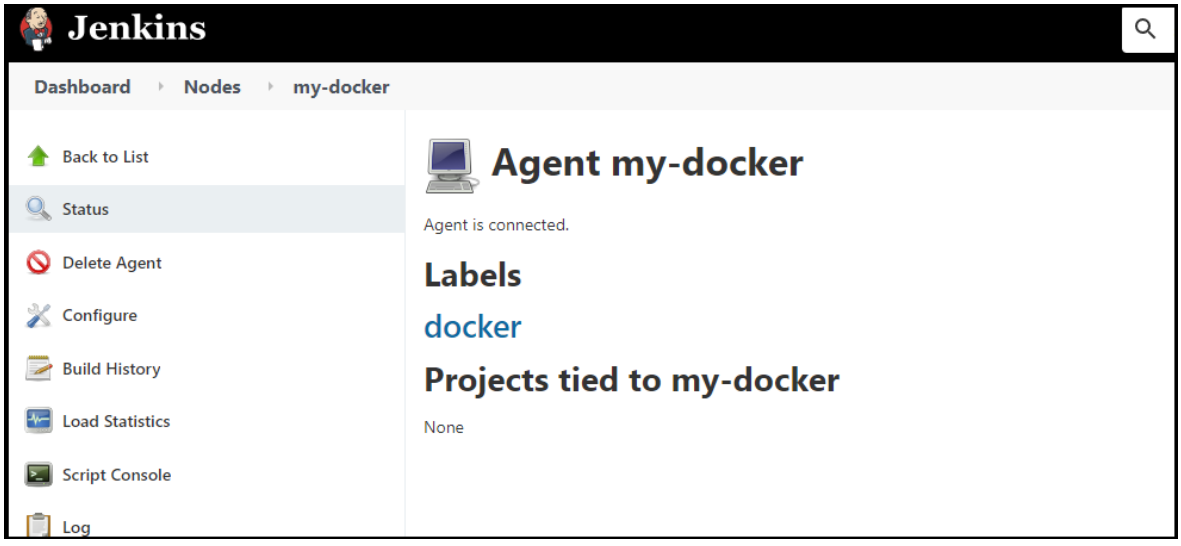
2. Use az container create to create and Azure Container Instance. Replace the placeholders with the values obtained when you created the work agent.

Azure CLI

```
az container create \
  --name my-dock \
  --resource-group my-resourcegroup \
  --ip-address Public --image jenkins/inbound-agent:latest \
  --os-type linux \
  --ports 80 \
  --command-line "jenkins-agent -url http://jenkinsserver:port
<JENKINS_SECRET> <AGENT_NAME>"
```

Replace `http://jenkinsserver:port`, `<JENKINS_SECRET>`, and `<AGENT_NAME>` with your Jenkins controller and agent information. After the container starts, it will connect to the Jenkins controller server automatically.

3. Return to the Jenkins dashboard and check the agent status.

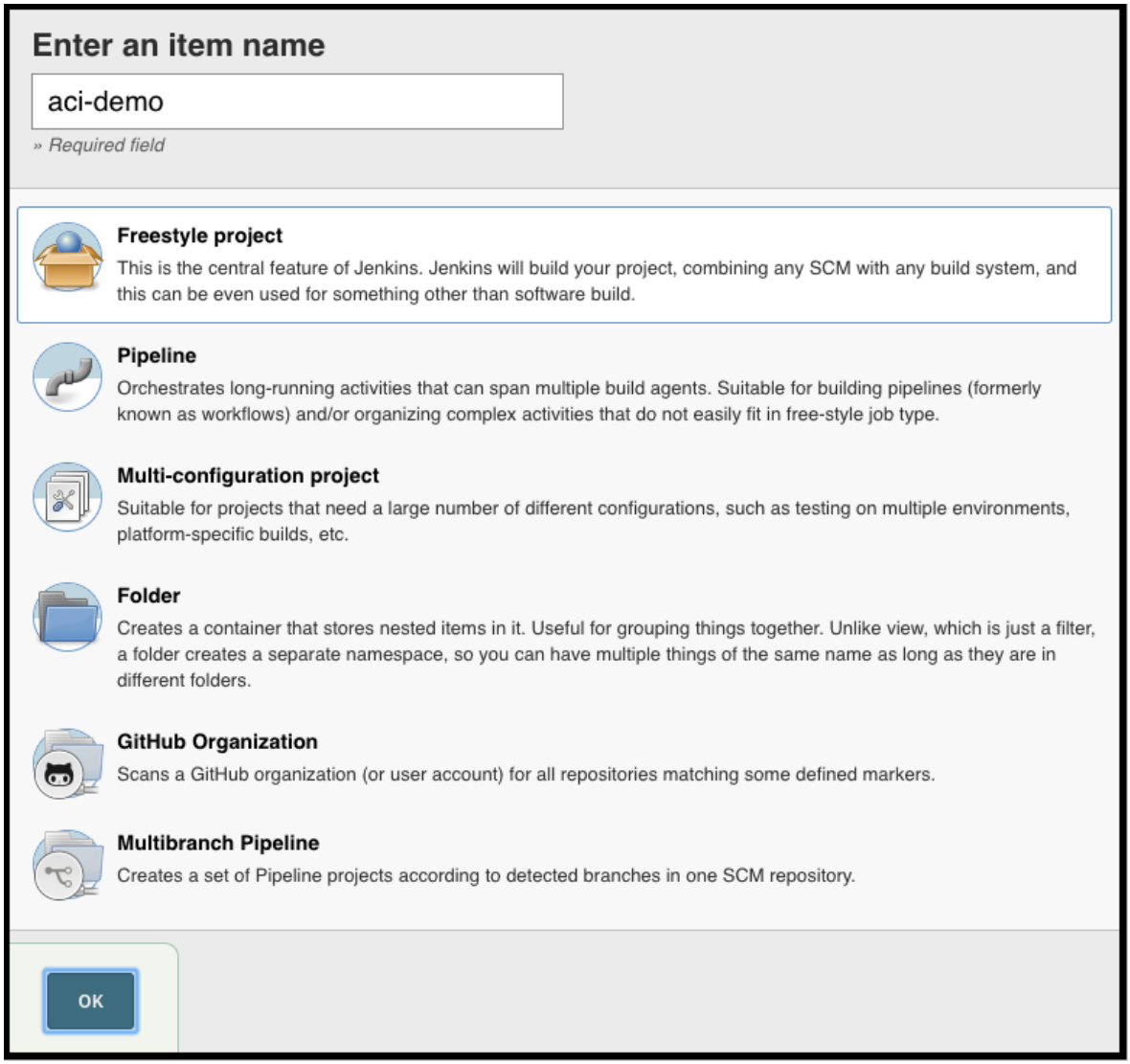


> ⓘ Note
>
> Jenkins agents connect to the controller via port `5000`, ensure that port is allowed inbound to the Jenkins Controller.

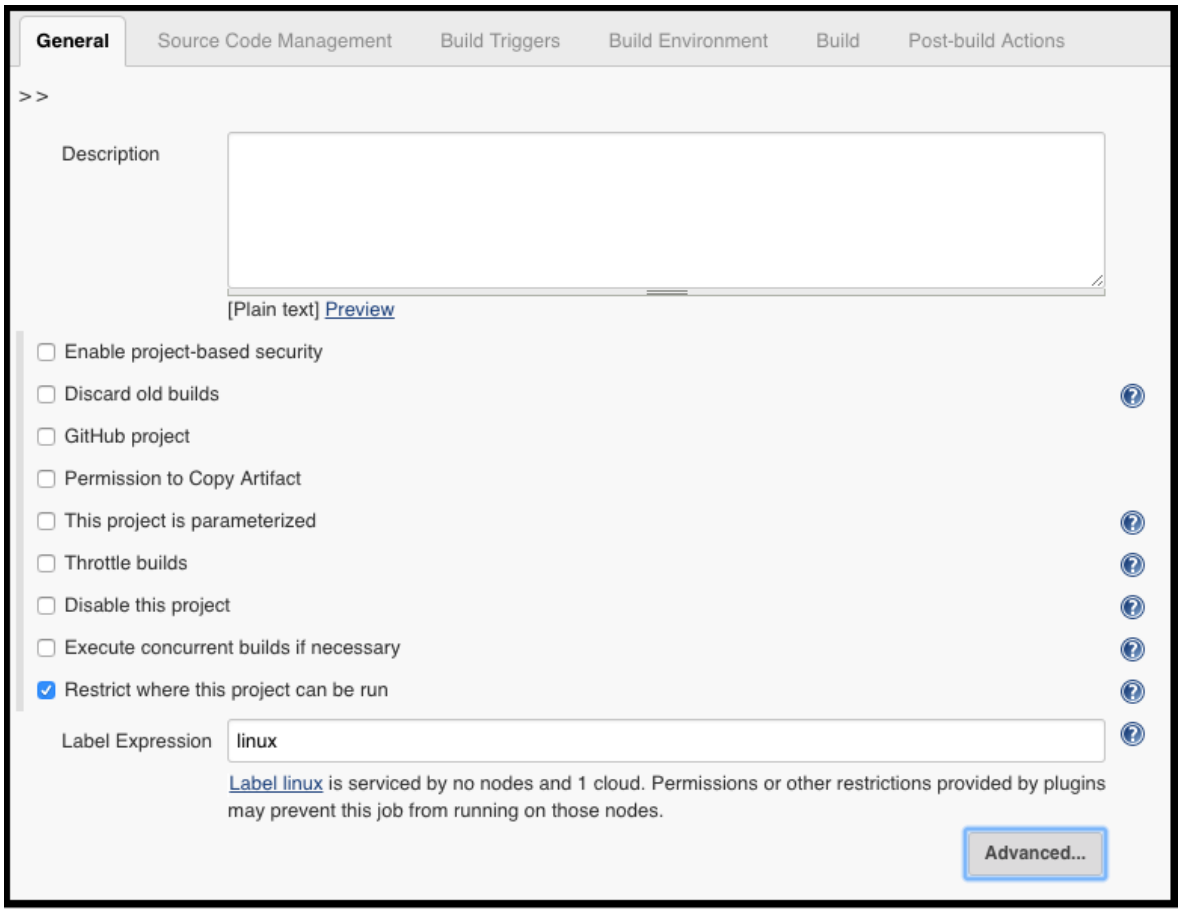
Create a build job

Now, a Jenkins build job is created to demonstrate Jenkins builds on an Azure container instance.

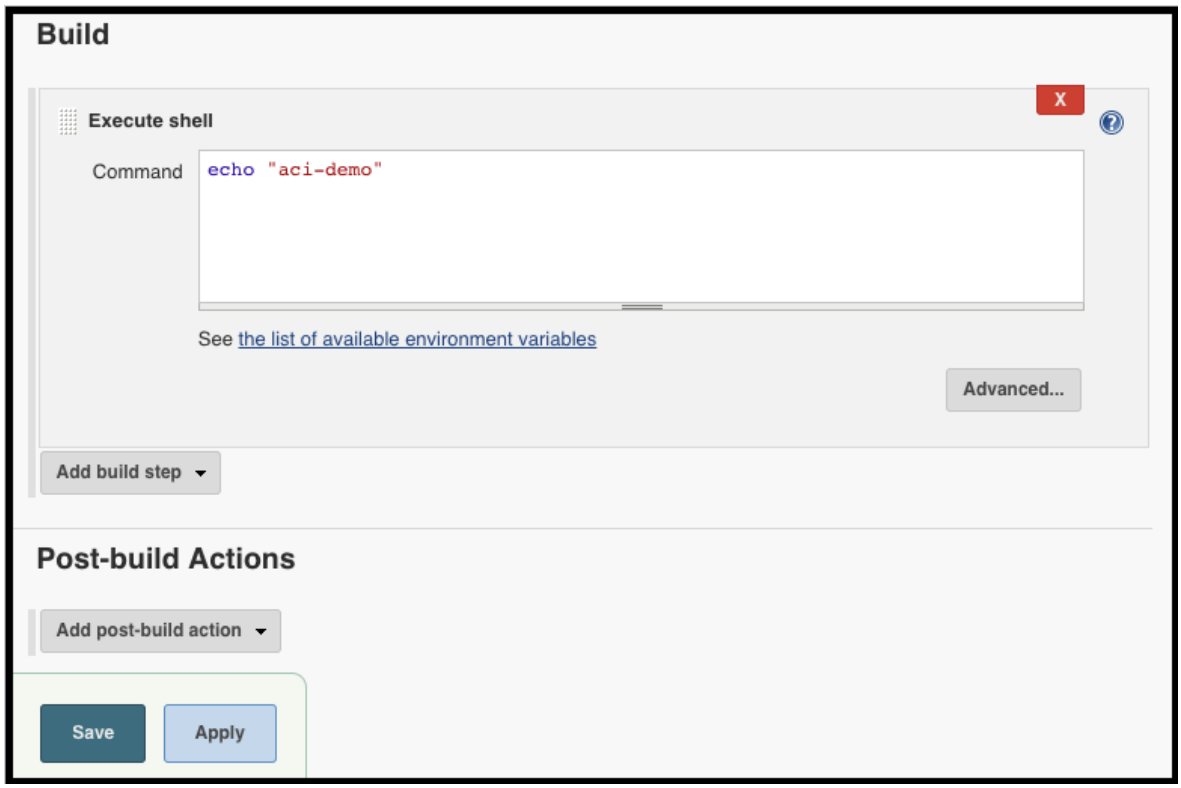
1. Select **New Item**, give the build project a name such as **aci-demo**, select **Freestyle project**, and select **OK**.



2. Under **General**, ensure that **Restrict where this project can be run** is selected. Enter **linux** for the label expression. This configuration ensures that this build job runs on the ACI cloud.



3. Under **Build**, select **Add build step** and select **Execute Shell**. Enter `echo "aci-demo"` as the command.

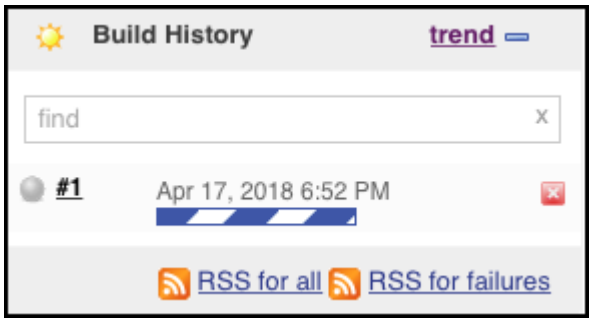


4. Select **Save**.

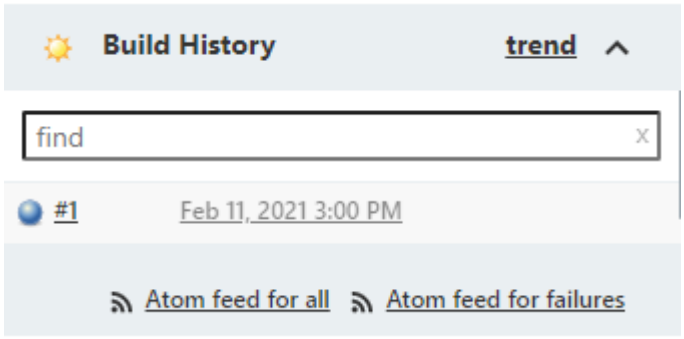
Run the build job

To test the build job and observe Azure Container Instances manually start a build.

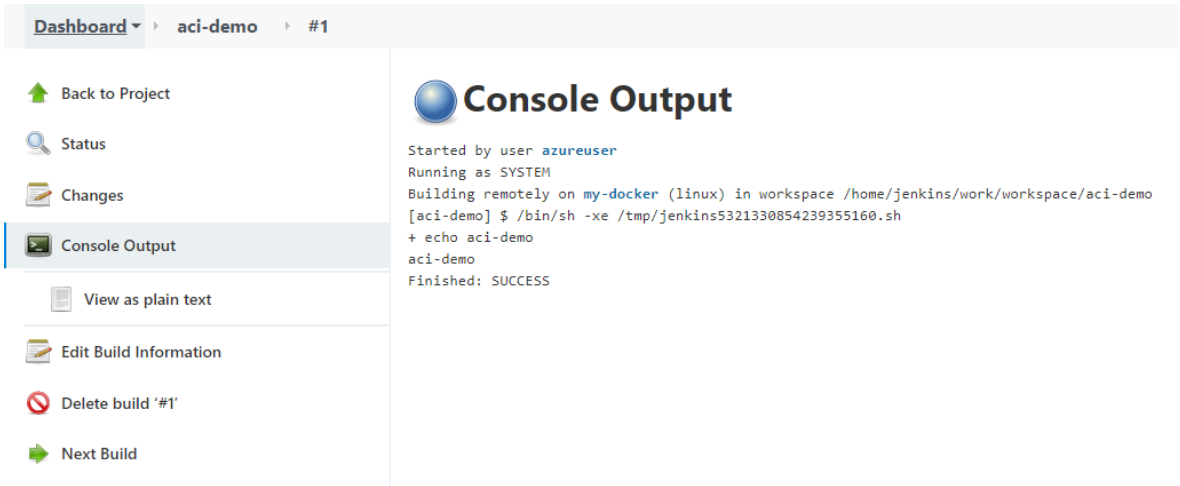
1. Select **Build Now** to start a build job. Once the job starts you'll see a status that's similar to the following image:



2. Click build **#1** in the **Build History**.



3. Select **Console Output** to view the builds output.



Next steps

CI/CD to Azure App Service

Tutorial: Scale Jenkins deployments with VM running in Azure

Article • 03/24/2022

> **ⓘ Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

This tutorial shows how to create a Linux virtual machines in Azure and add the VM as a work node to Jenkins.

In this tutorial, you will:

- ✓ Create agent machine
- ✓ Add agent to Jenkins
- ✓ Create a new Jenkins freestyle job
- ✓ Run the job on an Azure VM agent

Prerequisites

- **Jenkins installation**: If you don't have access to a Jenkins installation, configure Jenkins using Azure CLI

Configure agent virtual machine

1. Use az group create to create an Azure resource group.

   Azure CLI

   ```
   az group create --name <resource_group> --location <location>
   ```

2. Use az vm create to create a virtual machine.

   Azure CLI

   ```
   az vm create --resource-group <resource-group> --name <vm_name> --image
   UbuntuLTS --admin-username azureuser --admin-password "<password>"
   ```

Key points:

- You can also upload your ssh key with the following command `--ssh-key-value <ssh_path>`.

3. Install the JDK.

Windows

a. Log in to the virtual machine using an SSH tool or using Remote Desktop Connection.

b. Download the JDK that is appropriate for your environment.

c. Install JDK

Configure Jenkins URL

If you use JNLP, you'll need to configure the Jenkins URL.

1. From the menu, select **Manage Jenkins**.
2. Under **System Configuration**, select **Configure System**.
3. Verify that the **Jenkins URL** is set to the HTTP address of your Jenkins installation - `http://<your_host>.<your_domain>:8080/`.
4. Select **Save**.

Add agent to Jenkins

1. From the menu, select **Manage Jenkins**.
2. Under **System Configuration**, select **Manage Nodes and Clouds**.
3. From the menu, select **New Node**.
4. Enter a value for **Node Name**.
5. Select **Permanent Agent**.
6. Select **OK**.

7. Specify values for the following fields:

   - **Name**: Specify a unique name that identifies an agent within the new Jenkins installation. This value can be different from the agent hostname. However, it's convenient to make them the two values the same. The name value is allowed any special character from the following list: `?*/\%!@#$^&|<>[]:;`.

   - **Remote root directory**: An agent needs to have a directory dedicated to Jenkins. Specify the path to this directory on the agent. It is best to use an absolute path, such as `/home/azureuser/work` or `c:\jenkins`. This should be a path local to the agent machine. There is no need for this path to be visible from the master. If you use a relative path, such as ./jenkins-agent, the path will be relative to the working directory provided by the Launch method.

   - **Labels**: Labels are used to group semantically related agents into one logical group. For example, you could define a label of `UBUNTU` for all your agents running the Ubuntu distro of Linux.

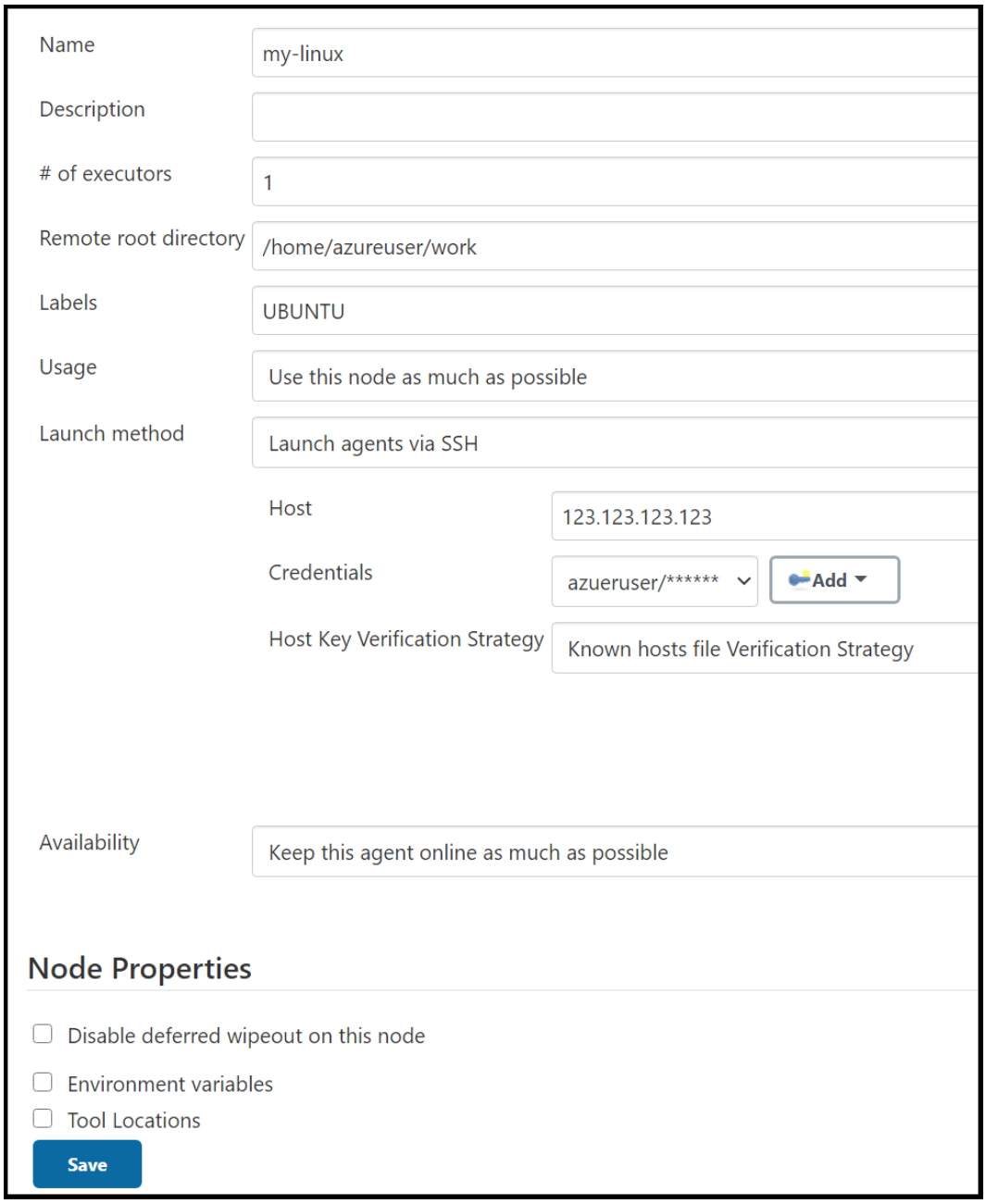
   - **Launch method**: There are two options to start the remote Jenkins node: **Launch agents via SSH** and **Launch agent via execution of command on the master**:

     - **Launch agents via SSH**: Specify the values for the following fields:

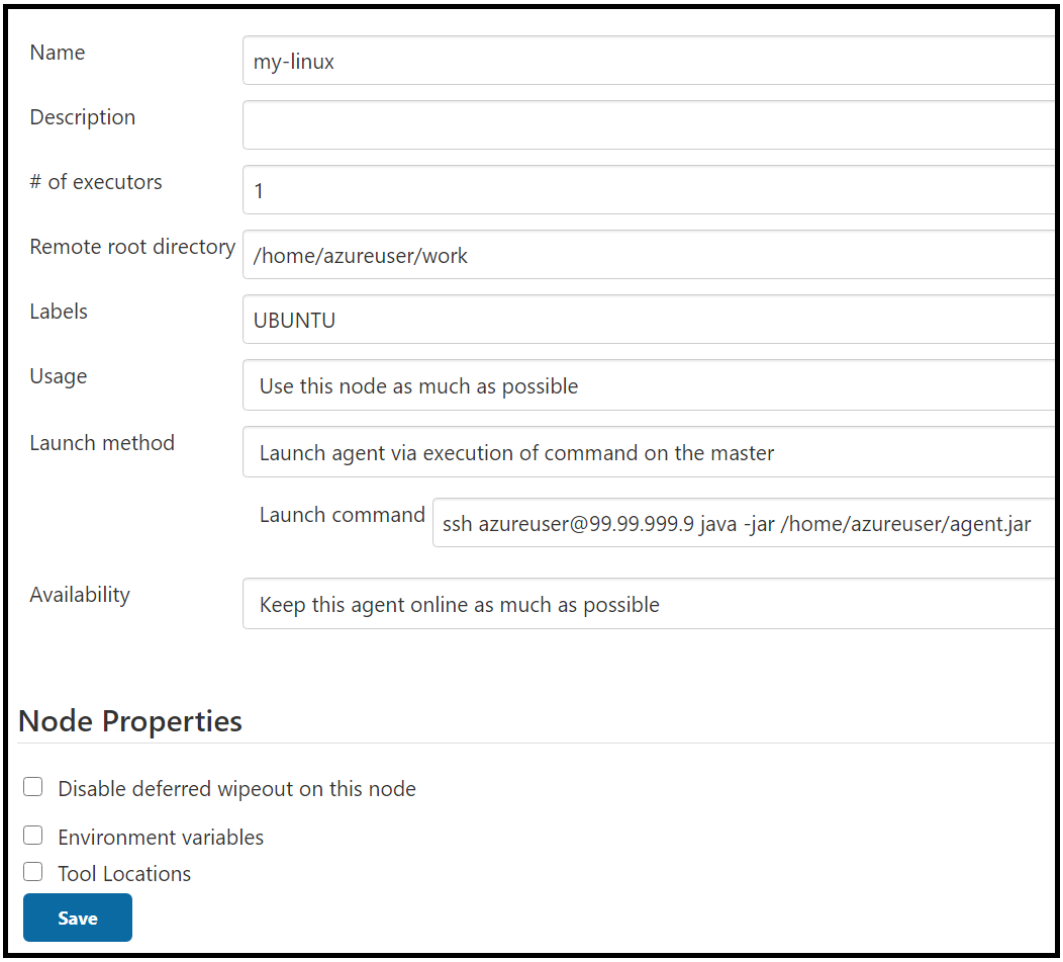
       - **Host**: VM public IP address or domain name. For example, `123.123.123.123` or `example.com`

       - **Credentials**: Select a credential to be used for logging in to the remote host. You can also select the **Add** button to define a new credential and then select that new credential once it's create.

       - **Host Key Verification Strategy**: Controls how Jenkins verifies the SSH key presented by the remote host whilst connecting.



- **Launch agent via execution of command on the master**:
  - Download the `agent.jar` from `https://<your_jenkins_host_name>/jnlpJars/agent.jar`. For example, `https://localhost:8443/jnlpJars/agent.jar`.
  - Upload `agent.jar` to your virtual machine
  - Start Jenkins with command `ssh <node_host> java -jar <remote_agentjar_path>`. For example, `ssh azureuser@99.99.999.9 java -jar /home/azureuser/agent.jar`.



8. Select **Save**.

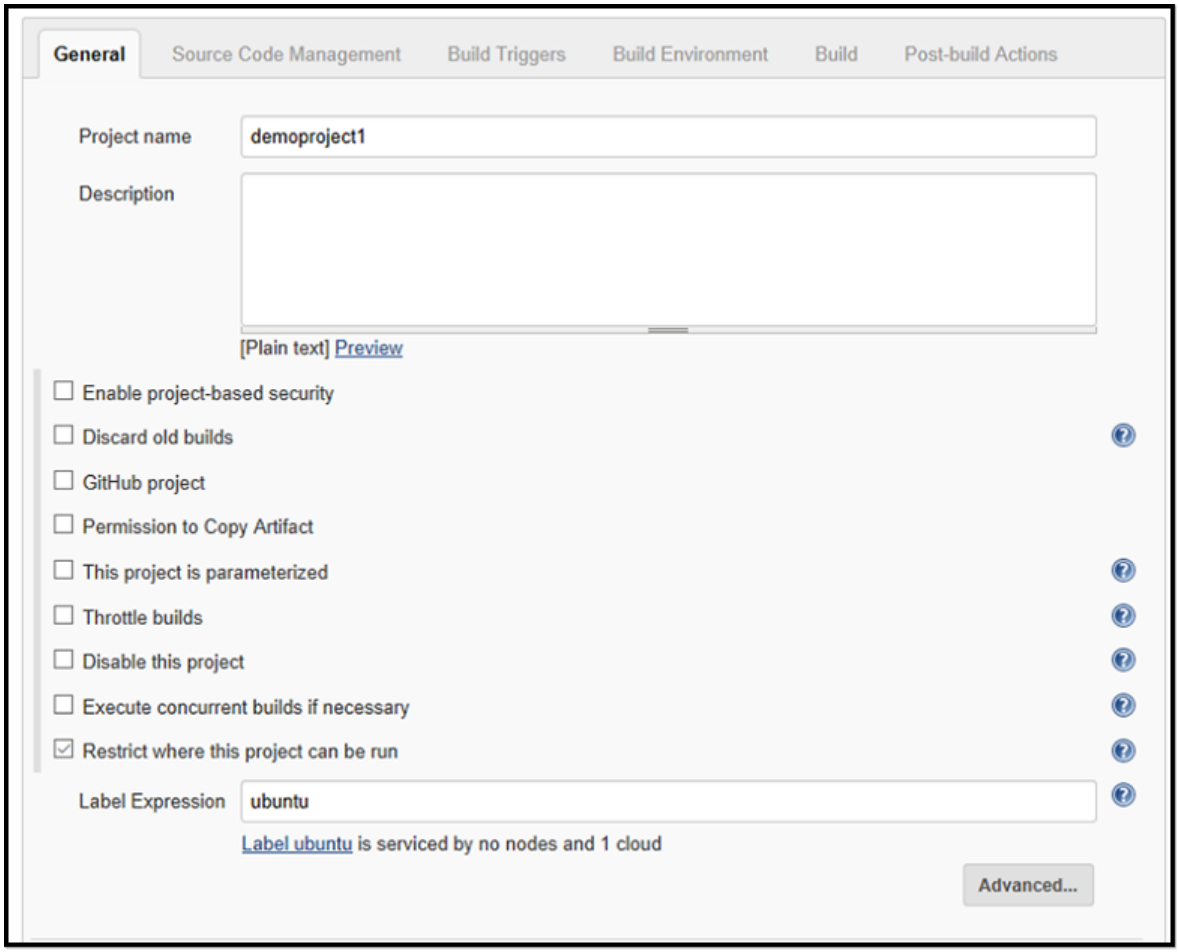
After you define the configurations, Jenkins adds the virtual machine as a new work node.

```
<===[JENKINS REMOTING CAPACITY]===>channel started
Remoting version: 3.36.1
This is a Unix agent
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by jenkins.slaves.StandardOutputSwapper$ChannelSwapper to constructor java.io.FileDescriptor(int)
WARNING: Please consider reporting this to the maintainers of jenkins.slaves.StandardOutputSwapper$ChannelSwapper
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Evacuated stdout
Agent successfully connected and online
```

Create a job in Jenkins

1. From the menu, select **New Item**.
2. Enter `demoproject1` for the name.
3. Select **Freestyle project**.
4. Select **OK**.
5. In the **General** tab, choose **Restrict where project can be run** and type `ubuntu` in **Label Expression**. You see a message confirming that the label is served by the

cloud configuration created in the previous step.



6. In the **Source Code Management** tab, select **Git** and add the following URL into the **Repository URL** field: `https://github.com/spring-projects/spring-petclinic.git`

7. In the **Build** tab, select **Add build step**, then **Invoke top-level Maven targets**. Enter `package` in the **Goals** field.

8. Select **Save**.

Build the new job on an Azure VM agent

1. Select the job you created in the previous step.

2. Select **Build now**. A new build is queued, but doesn't start until an agent VM is created in your Azure subscription.

3. Once the build is complete, go to **Console output**. You see that the build was performed remotely on an Azure agent.

Next steps

CI/CD to Azure App Service

Tutorial: Deploy apps to Azure Spring Apps using Jenkins and the Azure CLI

Article • 05/30/2023

> **Note**
>
> Azure Spring Apps is the new name for the Azure Spring Cloud service. Although the service has a new name, you'll see the old name in some places for a while as we work to update assets such as screenshots, videos, and diagrams.

Azure Spring Apps is a fully managed microservice development with built-in service discovery and configuration management. The service makes it easy to deploy Spring Boot-based microservice applications to Azure. This tutorial demonstrates how you can use Azure CLI in Jenkins to automate continuous integration and delivery (CI/CD) for Azure Spring Apps.

In this tutorial, you'll complete these tasks:

- ✓ Provision a service instance and launch a Java Spring application
- ✓ Prepare your Jenkins server
- ✓ Use the Azure CLI in a Jenkins pipeline to build and deploy the microservice applications

Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free account before you begin.
- **Jenkins**: Install Jenkins on a Linux VM
- **GitHub account**: If you don't have a GitHub account, create a free account before you begin.

Provision a service instance and launch a Java Spring application

We use Piggy Metrics as the sample Microsoft service application and follow the same steps in Quickstart: Launch a Java Spring application using the Azure CLI to provision the service instance and set up the applications. If you've already gone through the same

process, you can skip to the next section. Otherwise, included in the following are the Azure CLI commands. Refer to Quickstart: Launch a Java Spring application using the Azure CLI to get more information.

Your local machine needs to meet the same prerequisite as the Jenkins build server. Make sure the following are installed to build and deploy the microservice applications:

- Git
- JDK 8
- Maven 3.0 or above
- Azure CLI installed, version 2.0.67 or higher

1. Install the Azure Spring Apps extension:

   Azure CLI

   ```
   az extension add --name spring
   ```

2. Create a resource group to contain your Azure Spring Apps service:

   Azure CLI

   ```
   az group create --location eastus --name <resource group name>
   ```

3. Provision an instance of Azure Spring Apps:

   Azure CLI

   ```
   az spring create -n <service name> -g <resource group name>
   ```

4. Fork the Piggy Metrics repo to your own GitHub account. In your local machine, clone your repo in a directory called `source-code`:

   Bash

   ```
   mkdir source-code
   git clone https://github.com/<your GitHub id>/piggymetrics
   ```

5. Set up your configuration server. Make sure you replace <your GitHub id> with the correct value.

   Azure CLI

```
az spring config-server git set -n <your-service-name> --uri
https://github.com/<your GitHub id>/piggymetrics --label config
```

6. Build the project:

```Bash
cd piggymetrics
mvn clean package -D skipTests
```

7. Create the three microservices: **gateway**, **auth-service**, and **account-service**:

```Azure CLI
az spring app create --n gateway -s <service name> -g <resource group
name>
az spring app create --n auth-service -s <service name> -g <resource
group name>
az spring app create --n account-service -s <service name> -g <resource
group name>
```

8. Deploy the applications:

```Azure CLI
az spring app deploy -n gateway -s <service name> -g <resource group
name> --jar-path ./gateway/target/gateway.jar
az spring app deploy -n account-service -s <service name> -g <resource
group name> --jar-path ./account-service/target/account-service.jar
az spring app deploy -n auth-service -s <service name> -g <resource
group name> --jar-path ./auth-service/target/auth-service.jar
```

9. Assign public endpoint to gateway:

```Azure CLI
az spring app update -n gateway -s <service name> -g <resource group
name> --is-public true
```

10. Query the gateway application to get the url so that you can verify that the application is running.

```Azure CLI
az spring app show --name gateway | grep url
```

11. Navigate to the URL provided by the previous command to run the PiggyMetrics application.

Prepare Jenkins server

In this section, you prepare the Jenkins server to run a build, which is fine for testing. However, because of security implication, you should use an Azure VM agent or Azure Container agent to spin up an agent in Azure to run your builds.

Install plug-ins

1. Log in to your Jenkins server.
2. Select **Manage Jenkins**.
3. Select **Manage Plugins**.
4. On the **Available** tab, select the following plug-ins:

   - GitHub Integration
   - Azure Credential

   If these plug-ins don't appear in the list, check the **Installed** tab to see if they're already installed.
5. To install the plug-ins, select **Download now and install after restart**.
6. Restart your Jenkins server to complete the installation.

Add your Azure Service Principal credential in Jenkins credential store

1. You need an Azure Service Principal to deploy to Azure. For more information, see the Create service principal section in the Deploy to Azure App Service tutorial. The output from `az ad sp create-for-rbac` looks something like this:

```
{
    "appId": "xxxxxx-xxx-xxxx-xxx-xxxxxxxxxxxx",
    "displayName": "xxxxxxxjenkinssp",
    "name": "http://xxxxxxxjenkinssp",
    "password": "xxxxxx-xxx-xxxx-xxx-xxxxxxxxxxxx",
    "tenant": "xxxxxx--xxx-xxxx-xxx-xxxxxxxxxxxx"
}
```

2. On the Jenkins dashboard, select **Credentials** > **System**. Then, select **Global credentials(unrestricted)**.

3. Select **Add Credentials**.

4. Select **Microsoft Azure Service Principal** as kind.

5. Supply values for the following fields:

   - **Subscription ID**: Azure subscription ID
   - **Client ID**: Service principal appid
   - **Client Secret**: Service principal password
   - **Tenant ID**: Microsoft account tenant ID
   - **Azure Environment**: Select the appropriate value for your environment. For example, use **Azure** for Azure Global
   - **ID**: Set as `azure_service_principal`. We use this ID in a later step in this article
   - **Description**: This value is optional, but recommended from a documentation/maintenance standpoint.

Install Maven and Azure CLI spring extension

The sample pipeline uses Maven to build and Azure CLI to deploy to the service instance. When Jenkins is installed, it creates an admin account named *jenkins*. Ensure that the user *jenkins* has permission to run the spring extension.

1. Connect to the Jenkins controller via SSH.

2. Install Maven.

```Bash
sudo apt-get install maven
```

3. Verify that the Azure CLI is installed by entering `az version`. If the Azure CLI isn't installed, see Installing the Azure CLI.

4. Switch to the `jenkins` user:

```Bash
sudo su jenkins
```

5. Install the spring extension:

Azure CLI

```
az extension add --name spring
```

Create a Jenkinsfile

1. In your own repo - `https://github.com/your_github_id/piggymetrics` - create a **Jenkinsfile** in the root.

2. Update the file as follows. Make sure you replace the values of **<resource group name>** and **<service name>**. Replace **azure_service_principal** with the right ID if you use a different value when you added the credential in Jenkins.

groovy

```
    node {
      stage('init') {
        checkout scm
      }
      stage('build') {
        sh 'mvn clean package'
      }
      stage('deploy') {

withCredentials([azureServicePrincipal('azure_service_principal')]) {
          // Log in to Azure
          sh '''
            az login --service-principal -u $AZURE_CLIENT_ID -p
$AZURE_CLIENT_SECRET -t $AZURE_TENANT_ID
            az account set -s $AZURE_SUBSCRIPTION_ID
          '''
          // Set default resource group name and service name. Replace
<resource group name> and <service name> with the right values
          sh 'az config set defaults.group=<resource group name>'
          sh 'az config set defaults.spring=<service name>'

          // Deploy applications
          sh 'az spring app deploy -n gateway --jar-path
./gateway/target/gateway.jar'
          sh 'az spring app deploy -n account-service --jar-path
./account-service/target/account-service.jar'
          sh 'az spring app deploy -n auth-service --jar-path ./auth-
service/target/auth-service.jar'
          sh 'az logout'
        }
      }
    }
```

3. Save and commit the change.

Create the job

1. On the Jenkins dashboard, select **New Item**.
2. Provide a name, *Deploy-PiggyMetrics* for the job and select **Pipeline**. Click OK.
3. Select the **Pipeline** tab.
4. For **Definition**, select **Pipeline script from SCM**.
5. For **SCM**, select **Git**.
6. Enter the GitHub URL for your forked repo: `https://github.com/<your GitHub id>/piggymetrics.git`.
7. For **Branch Specifier (black for 'any')**, select **/Azure**.
8. For **Script path**, select **Jenkinsfile**.
9. Select **Save**

Validate and run the job

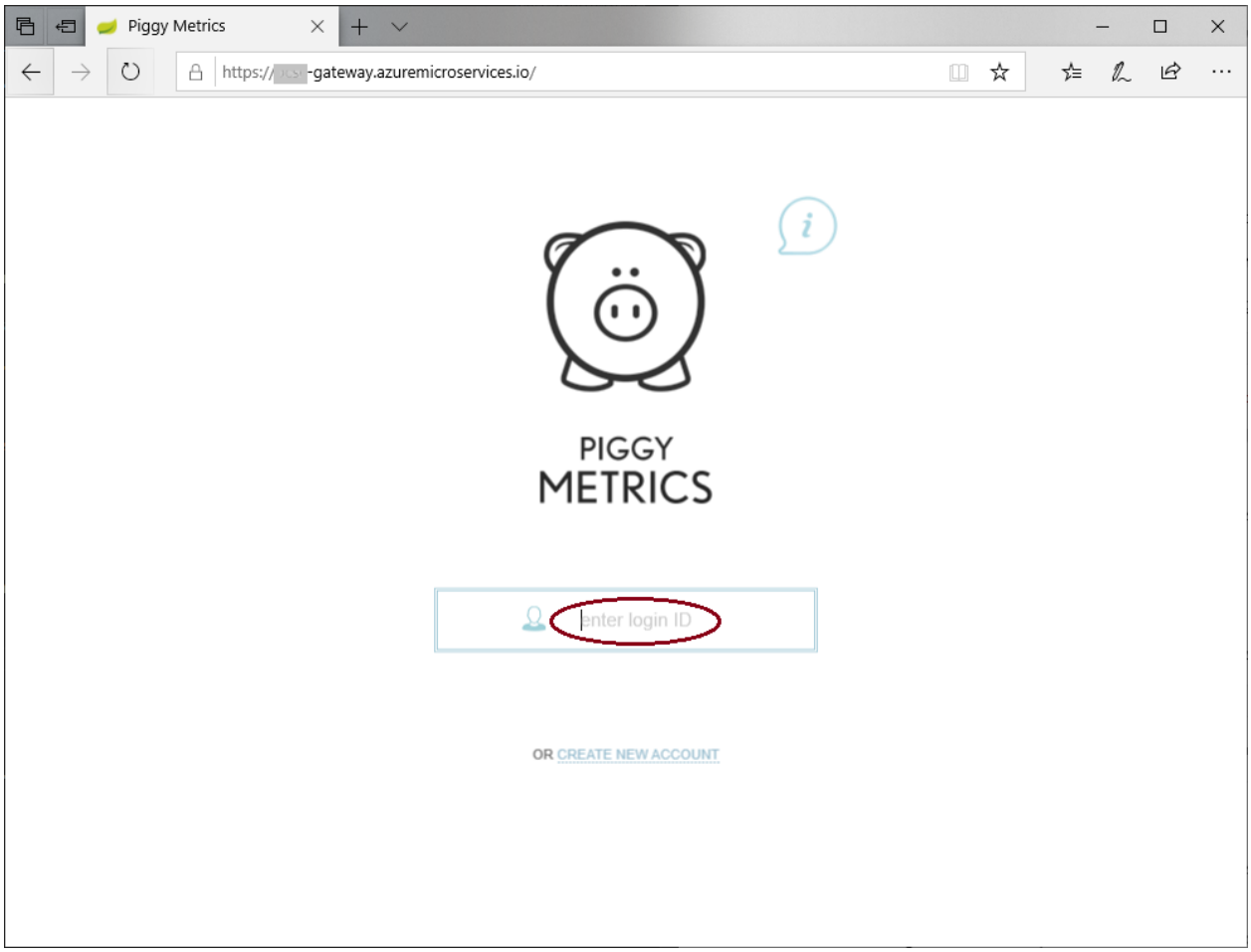
Before running the job, edit the text in the login input box to **enter login ID**.

1. In your repo, open `index.html` in `/gateway/src/main/resources/static/`.
2. Search for `enter your login` and update that text to `enter login ID`.

```HTML
<input class="frontforms" id="frontloginform" name="username"
placeholder="enter login ID" type="text" autocomplete="off"/>
```

3. Save and commit the change.
4. Run the job in Jenkins manually. On the Jenkins dashboard, select the job `Deploy-PiggyMetrics` and then select **Build Now**.

After the job is complete, navigate to the public IP of the `gateway` application and verify that your application has been updated.



Clean up resources

When no longer needed, delete the resources created in this article:

Azure CLI

```
az group delete -y --no-wait -n <resource group name>
```

Next steps

Jenkins on Azure

Tutorial: Deploy from GitHub to Azure Kubernetes Service using Jenkins

Article • 03/24/2022

> **ⓘ Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

This tutorial deploys a sample app from GitHub to an Azure Kubernetes Service (AKS) cluster by setting up continuous integration (CI) and continuous deployment (CD) in Jenkins.

In this tutorial, you'll complete these tasks:

- ✓ Deploy a sample Azure vote app to an AKS cluster.
- ✓ Create a basic Jenkins project.
- ✓ Set up credentials for Jenkins to interact with ACR.
- ✓ Create a Jenkins build job and GitHub webhook for automated builds.
- ✓ Test the CI/CD pipeline to update an application in AKS based on GitHub code commits.

Prerequisites

To complete this tutorial, you need these items:

- Basic understanding of Kubernetes, Git, CI/CD, and container images
- An AKS cluster and `kubectl` configured with the AKS cluster credentials.
- An Azure Container Registry (ACR) registry, the ACR login server name, and the AKS cluster configured to authenticate with the ACR registry.
- A Jenkins Controller Deployed to an Azure Virtual Machine.
- The Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.
- Docker installed on your development system

- A GitHub account, GitHub personal access token, and Git client installed on your development system
- If you provide your own Jenkins instance rather than this sample scripted way to deploy Jenkins, your Jenkins instance needs Docker installed and configured and kubectl.

Prepare your app

In this article, you use a sample Azure vote application that contains a web interface and Redis for temporary data storage.

Before you integrate Jenkins and AKS for automated deployments, first manually prepare and deploy the Azure vote application to your AKS cluster. This manual deployment lets you see the application in action.

> **Note**
>
> The sample Azure vote application uses a Linux pod that is scheduled to run on a Linux node. The flow outlined in this article also works for a Windows Server pod scheduled on a Windows Server node.

Fork the following GitHub repository for the sample application - https://github.com/Azure-Samples/azure-voting-app-redis. To fork the repo to your own GitHub account, select the **Fork** button in the top right-hand corner.

Clone the fork to your development system. Make sure you use the URL of your fork when cloning this repo:

```
Console

git clone https://github.com/<your-github-account>/azure-voting-app-redis.git
```

Change to the directory of your cloned fork:

```
Console

cd azure-voting-app-redis
```

To create the container images needed for the sample application, use the *docker-compose.yaml* file with `docker-compose`:

Console

```
docker-compose up -d
```

The required base images are pulled and the application containers built. You can then use the docker images command to see the created image. Three images have been downloaded or created. The `azure-vote-front` image contains the application and uses the `nginx-flask` image as a base. The `redis` image is used to start a Redis instance:

```
$ docker images

REPOSITORY                    TAG        IMAGE ID            CREATED
SIZE
azure-vote-front              latest     9cc914e25834        40 seconds ago
694MB
redis                         latest     a1b99da73d05        7 days ago
106MB
tiangolo/uwsgi-nginx-flask    flask      788ca94b2313        9 months ago
694MB
```

Sign in to your Azure container registry.

Azure CLI

```
az acr login -n <acrLoginServer>
```

Replace `<acrLoginServer>` with your ACR login server.

Use the docker tag command to tag the image with the ACR login server name and a version number of `v1`. Use your own `<acrLoginServer>` name obtained in the previous step:

Console

```
docker tag azure-vote-front <acrLoginServer>/azure-vote-front:v1
```

Finally, push the *azure-vote-front* image to your ACR registry. Again, replace `<acrLoginServer>` with the login server name of your own ACR registry, such as `myacrregistry.azurecr.io`:

Console

```
docker push <acrLoginServer>/azure-vote-front:v1
```

Deploy the sample application to AKS

To deploy the sample application to your AKS cluster, you can use the Kubernetes manifest file in the root of the Azure vote repository repo. Open the `azure-vote-all-in-one-redis.yaml` manifest file with an editor such as `vi`. Replace `microsoft` with your ACR login server name. This value is found on line **60** of the manifest file:

YAML

```yaml
containers:
- name: azure-vote-front
  image: azuredocs/azure-vote-front
```

Next, use the kubectl apply command to deploy the application to your AKS cluster:

Console

```
kubectl apply -f azure-vote-all-in-one-redis.yaml
```

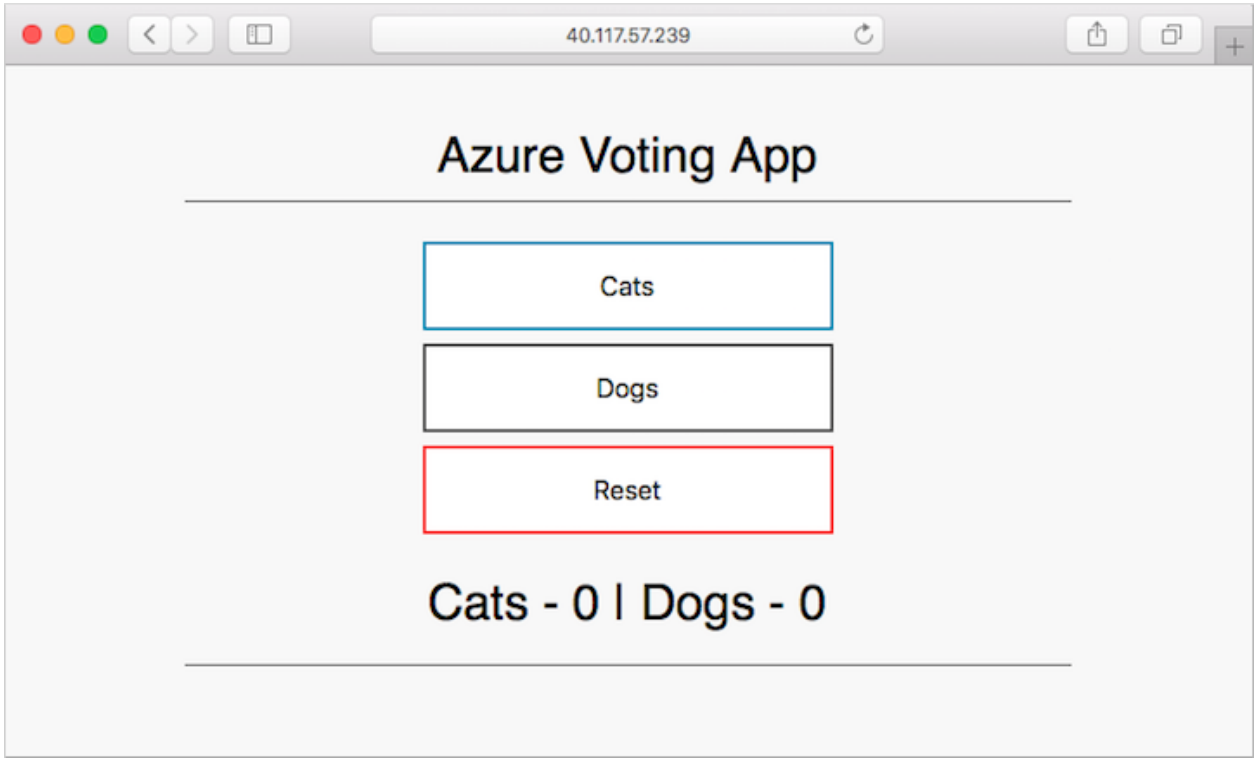
A Kubernetes load balancer service is created to expose the application to the internet. This process can take a few minutes. To monitor the progress of the load balancer deployment, use the kubectl get service command with the `--watch` argument. Once the *EXTERNAL-IP* address has changed from *pending* to an *IP address*, use `Control + C` to stop the kubectl watch process.

Console

```
$ kubectl get service azure-vote-front --watch

NAME               TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
azure-vote-front   LoadBalancer   10.0.215.27   <pending>     80:30747/TCP
22s
azure-vote-front   LoadBalancer   10.0.215.27   40.117.57.239   80:30747/TCP
2m
```

To see the application in action, open a web browser to the external IP address of your service. The Azure vote application is displayed, as shown in the following example:



Configure Jenkins Controller

Apply the following changes to enable AKS deployments from the Jenkins Controller:

Open port `80` inbound.

Azure CLI

```
az vm open-port \
--resource-group <Resource_Group_name> \
--name <Jenkins_Controller_VM>  \
--port 80 --priority 1020
```

Replace `<Resource_Group_name>` and `<Jenkins_Controller_VM>` with the appropriate values.

SSH into the Jenkins Controller

Azure CLI

```
ssh azureuser@<PublicIPAddres>
```

Replace `<PublicIPAddress>` with the IP address of the Jenkins Controller.

Install & Log into AzCLI

Azure CLI

```
curl -L https://aka.ms/InstallAzureCli | bash
```

Azure CLI

```
az login
```

> ⓘ **Note**
>
> To manually install AzCLI, follow these **instructions**.

Install Docker

Bash

```
sudo apt-get install apt-transport-https ca-certificates curl software-
properties-common -y;
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
-;
sudo apt-key fingerprint 0EBFCD88;
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable";
sudo apt-get update;
sudo apt-get install docker-ce -y;
```

Install Kubectl and Connect to AKS

Azure CLI

```
sudo az aks install-cli
sudo az aks get-credentials --resource-group <Resource_Group> --name
<AKS_Name>
```

Replace `<Resource_Group>` and `<AKS_Name>` with the appropriate values.

Configure access

Bash

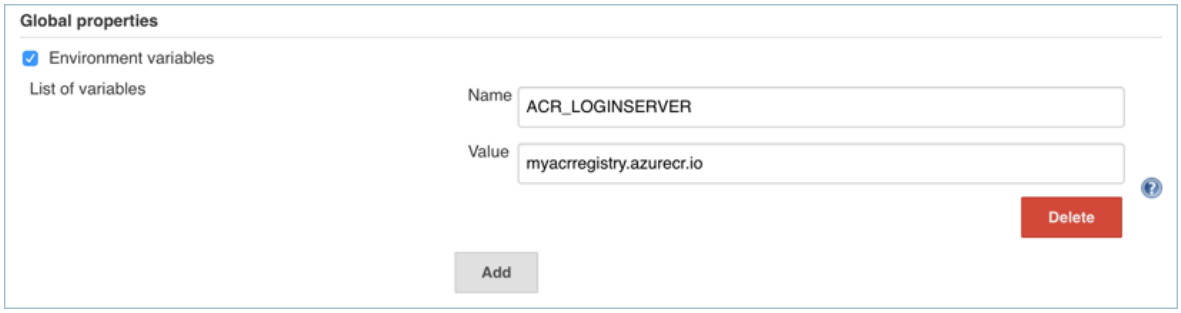
```
sudo usermod -aG docker jenkins;
sudo usermod -aG docker azureuser;
```

```
sudo touch /var/lib/jenkins/jenkins.install.InstallUtil.lastExecVersion;
sudo service jenkins restart;
sudo cp ~/.kube/config /var/lib/jenkins/.kube/
sudo chmod 777 /var/lib/jenkins/
sudo chmod 777 /var/lib/jenkins/config
```

Create a Jenkins environment variable

A Jenkins environment variable is used to hold the ACR login server name. This variable is referenced during the Jenkins build job. To create this environment variable, complete the following steps:

- On the left-hand side of the Jenkins portal, select **Manage Jenkins** > **Configure System**
- Under **Global Properties**, select **Environment variables**. Add a variable with the name `ACR_LOGINSERVER` and the value of your ACR login server.



- When complete, Select **Save** at the bottom of the page.

Create a Jenkins credential for ACR

During the CI/CD process, Jenkins builds new container images based on application updates, and needs to then *push* those images to the ACR registry.

To allow Jenkins to push updated container images to ACR, you need to specify credentials for ACR.

For separation of roles and permissions, configure a service principal for Jenkins with *Contributor* permissions to your ACR registry.

Create a service principal for Jenkins to use ACR

First, create a service principal using the az ad sp create-for-rbac command:

Azure CLI

```
az ad sp create-for-rbac
```

Bash

```
{
  "appId": "626dd8ea-042d-4043-a8df-4ef56273670f",
  "displayName": "azure-cli-2018-09-28-22-19-34",
  "name": "http://azure-cli-2018-09-28-22-19-34",
  "password": "1ceb4df3-c567-4fb6-955e-f95ac9460297",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db48"
}
```

Make a note of the *appId* and *password*. These values are used in following steps to configure the credential resource in Jenkins.

Get the resource ID of your ACR registry using the az acr show command, and store it as a variable.

Azure CLI

```
ACR_ID=$(az acr show --resource-group <Resource_Group> --name
<acrLoginServer> --query "id" --output tsv)
```

Replace `<Resource_Group>` and `<acrLoginServer>` with the appropriate values.

Create a role assignment to assign the service principal *Contributor* rights to the ACR registry.

Azure CLI

```
az role assignment create --assignee <appID> --role Contributor --scope
$ACR_ID
```

Replace `<appId>` with the value provided in the output of the pervious command use to create the service principal.

Create a credential resource in Jenkins for the ACR service principal

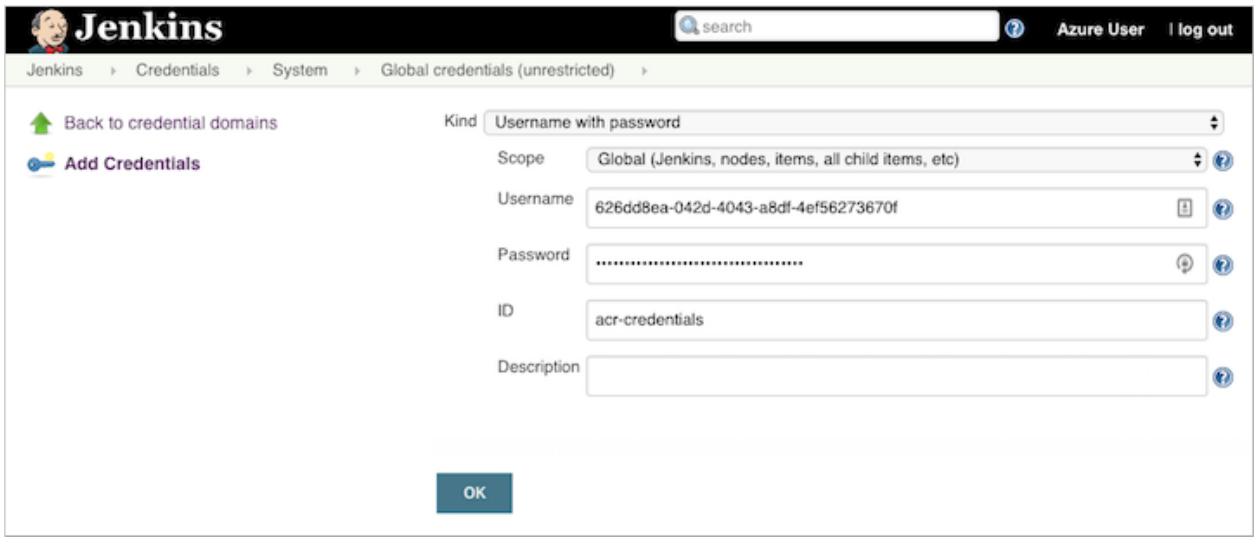
With the role assignment created in Azure, now store your ACR credentials in a Jenkins credential object. These credentials are referenced during the Jenkins build job.

Back on the left-hand side of the Jenkins portal, select **Manage Jenkins** > **Manage Credentials** > **Jenkins Store** > **Global credentials (unrestricted)** > **Add Credentials**

Ensure that the credential kind is **Username with password** and enter the following items:

- **Username** - The *appId* of the service principal created for authentication with your ACR registry.
- **Password** - The *password* of the service principal created for authentication with your ACR registry.
- **ID** - Credential identifier such as *acr-credentials*

When complete, the credentials form looks like the following example:



Select **OK** and return to the Jenkins portal.

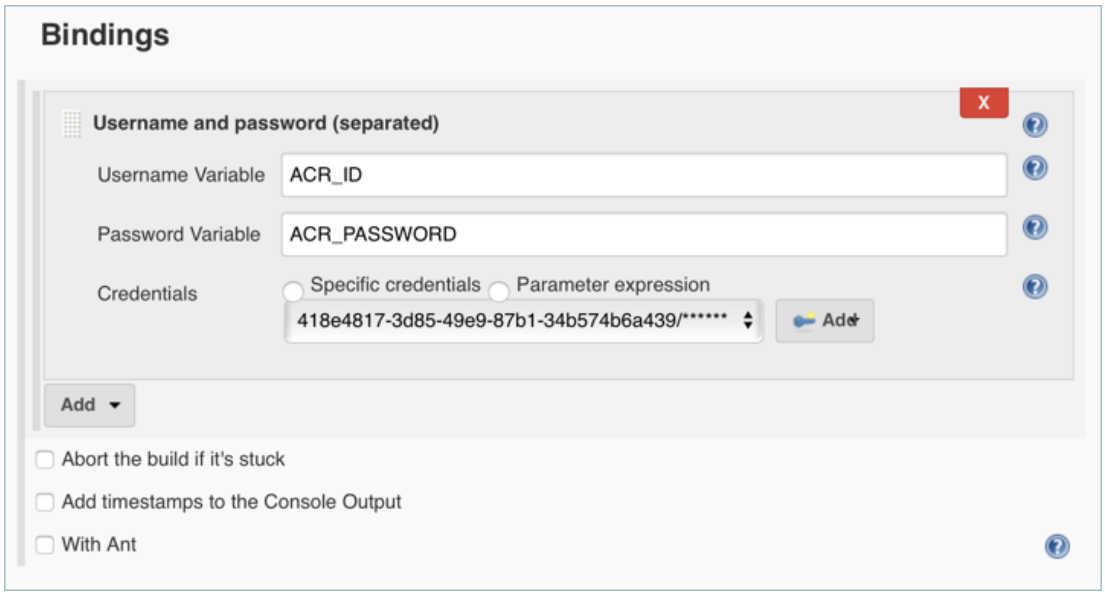
Create a Jenkins project

From the home page of your Jenkins portal, select **New item** on the left-hand side:

1. Enter *azure-vote* as job name. Choose **Freestyle project**, then select **OK**
2. Under the **General** section, select **GitHub project** and enter your forked repo URL, such as *https://github.com/<your-github-account>/azure-voting-app-redis*
3. Under the **Source code management** section, select **Git**, enter your forked repo `.git` URL, such as *https://github.com/<your-github-account>/azure-voting-app-redis.git*
4. Under the **Build Triggers** section, select **GitHub hook trigger for GITscm polling**
5. Under **Build Environment**, select **Use secret texts or files**

6. Under **Bindings**, select **Add** > **Username and password (separated)**

    - Enter `ACR_ID` for the **Username Variable**, and `ACR_PASSWORD` for the **Password Variable**



7. Choose to add a **Build Step** of type **Execute shell** and use the following text. This script builds a new container image and pushes it to your ACR registry.

```Bash
# Build new image and push to ACR.
WEB_IMAGE_NAME="${ACR_LOGINSERVER}/azure-vote-
front:kube${BUILD_NUMBER}"
docker build -t $WEB_IMAGE_NAME ./azure-vote
docker login ${ACR_LOGINSERVER} -u ${ACR_ID} -p ${ACR_PASSWORD}
docker push $WEB_IMAGE_NAME
```

8. Add another **Build Step** of type **Execute shell** and use the following text. This script updates the application deployment in AKS with the new container image from ACR.

```Bash
# Update kubernetes deployment with new image.
WEB_IMAGE_NAME="${ACR_LOGINSERVER}/azure-vote-
front:kube${BUILD_NUMBER}"
kubectl set image deployment/azure-vote-front azure-vote-
front=$WEB_IMAGE_NAME
```

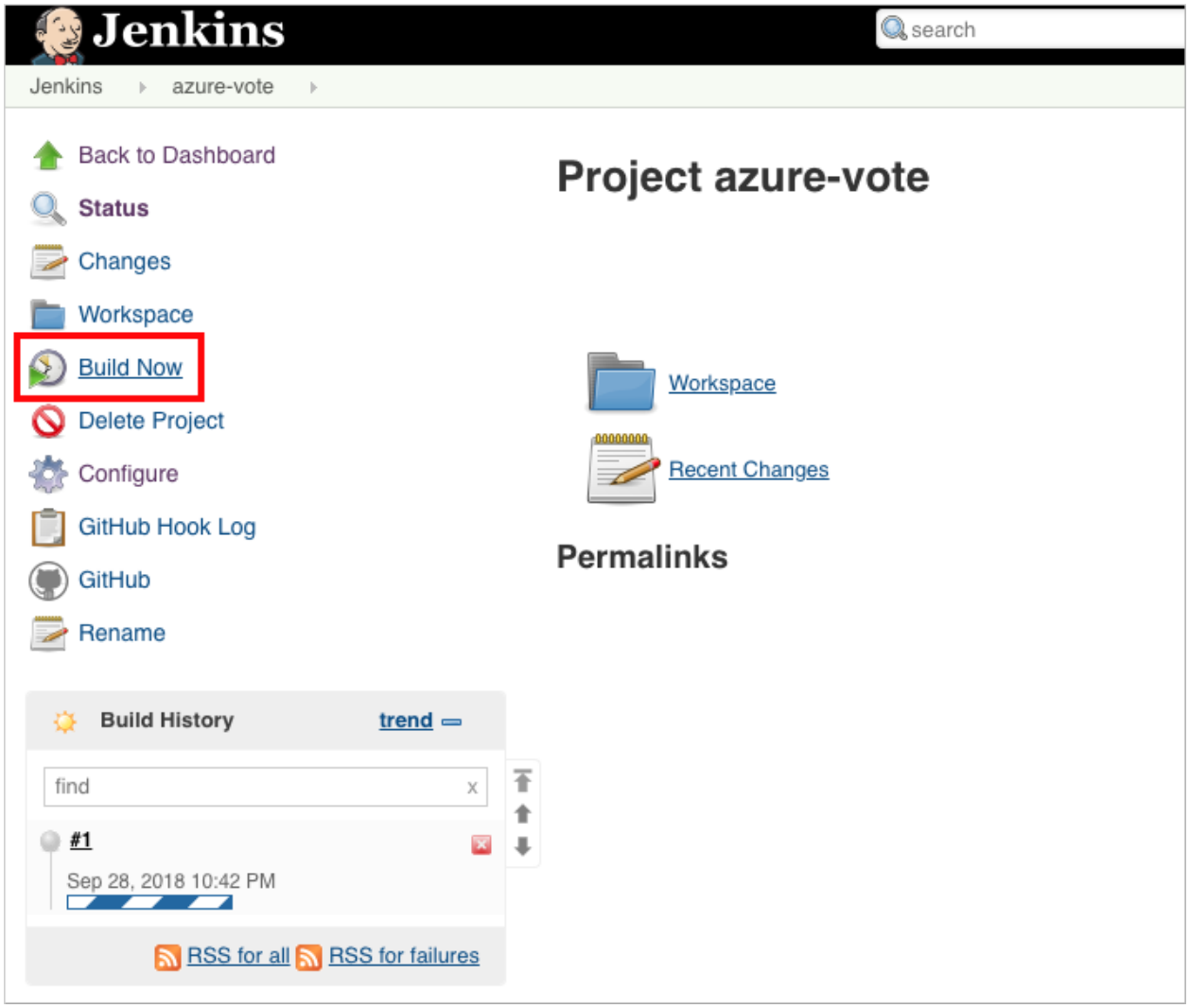
9. Once completed, click **Save**.

Test the Jenkins build

Before you automate the job based on GitHub commits, manually test the Jenkins build.

This build validates that the job has been correctly configured. It confirms the proper Kubernetes authentication file is in place, and that authentication to ACR working.

On the left-hand menu of the project, select **Build Now**.



The first build longer as the Docker image layers are pulled down to the Jenkins server.

The builds do the following tasks:

1. Clones the GitHub repository
2. Builds a new container image
3. Pushes the container image to the ACR registry
4. Updates the image used by the AKS deployment

Because no changes have been made to the application code, the web UI is unchanged.

Once the build job is complete, select **build #1** under build history. Select **Console Output** and view the output from the build process. The final line should indicate a

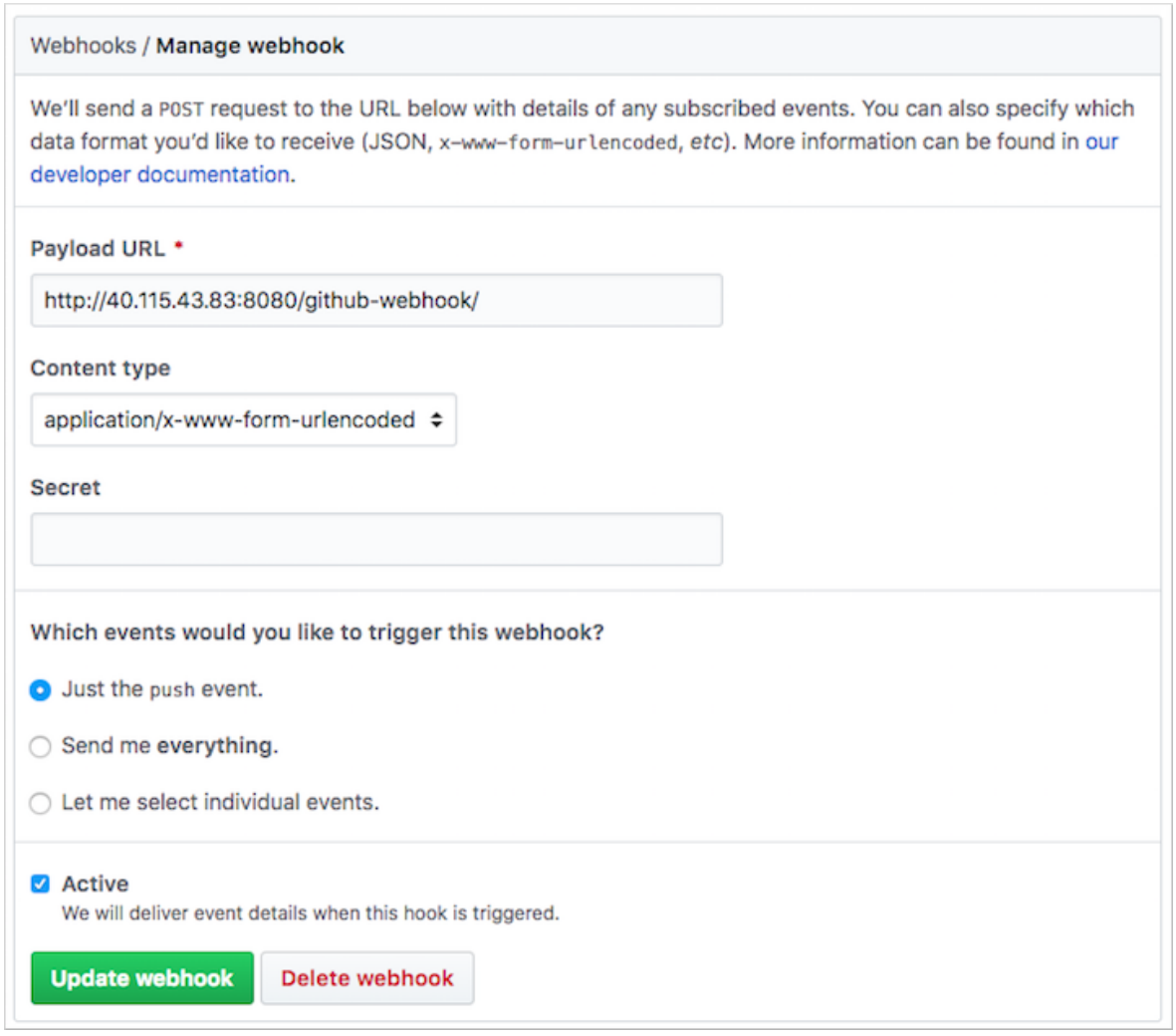
successful build.

Create a GitHub webhook

With a successful manual build complete, now integrate GitHub into the Jenkins build. Use a webhook to run the Jenkins build job each time code is committed to GitHub.

To create the GitHub webhook, complete the following steps:

1. Browse to your forked GitHub repository in a web browser.
2. Select **Settings**, then select **Webhooks** on the left-hand side.
3. Choose to **Add webhook**. For the *Payload URL*, enter `http://<publicIp:8080>/github-webhook/`, where `<publicIp>` is the IP address of the Jenkins server. Make sure to include the trailing `/`. Leave the other defaults for content type and to trigger on *push* events.
4. Select **Add webhook**.



Test the complete CI/CD pipeline

Now you can test the whole CI/CD pipeline. When you push a code commit to GitHub, the following steps happen:

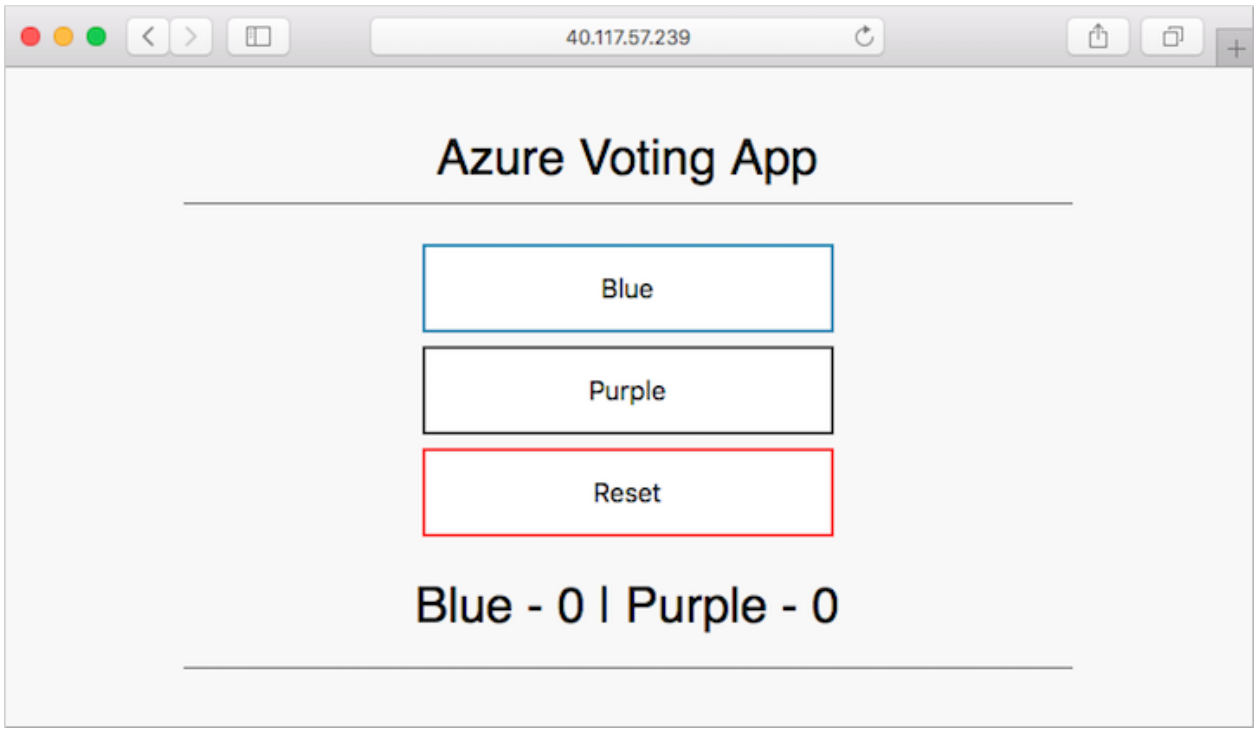
1. The GitHub webhook notifies Jenkins.
2. Jenkins starts the build job and pulls the latest code commit from GitHub.
3. A Docker build is started using the updated code, and the new container image is tagged with the latest build number.
4. This new container image is pushed to Azure Container Registry.
5. Your application running on Azure Kubernetes Service updates with the latest image from Azure Container Registry.

On your development machine, open up the cloned application with a code editor. Under the */azure-vote/azure-vote* directory, open the file named **config_file.cfg**. Update the vote values in this file to something other than cats and dogs, as shown in the following example:

```
# UI Configurations
TITLE = 'Azure Voting App'
VOTE1VALUE = 'Blue'
VOTE2VALUE = 'Purple'
SHOWHOST = 'false'
```

When updated, save the file, commit the changes, and push them to your fork of the GitHub repository. The GitHub webhook triggers a new build job in Jenkins. In the Jenkins web dashboard, monitor the build process. It takes a few seconds to pull the latest code, create and push the updated image, and deploy the updated application in AKS.

Once the build is complete, refresh your web browser of the sample Azure vote application. Your changes are displayed, as shown in the following example:



Next steps

Jenkins on Azure

Tutorial: Deploy to Azure App Service with Jenkins and the Azure CLI

Article • 05/30/2023

> **ⓘ Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

To deploy a Java web app to Azure, you can use Azure CLI in a Jenkins Pipeline. In this tutorial, you do the following tasks:

- ✓ Create a Jenkins VM
- ✓ Configure Jenkins
- ✓ Create a web app in Azure
- ✓ Prepare a GitHub repository
- ✓ Create Jenkins pipeline
- ✓ Run the pipeline and verify the web app

Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free account before you begin.
- **Jenkins** - Install Jenkins on a Linux VM
- **Azure CLI**: Install Azure CLI (version 2.0.67 or higher) on the Jenkins server.

Configure Jenkins

The following steps show how to install the required Java JDK and Maven on the Jenkins controller:

1. Sign in to Jenkins controller using SSH.
2. Download and install the Azul Zulu build of OpenJDK for Azure from an apt-get repository:

```
```

Bash

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
0xB1998361219BD9C9
sudo apt-add-repository "deb http://repos.azul.com/azure-only/zulu/apt
stable main"
sudo apt-get -q update
sudo apt-get -y install zulu-8-azure-jdk
```

3. Run the following command to install Maven:

Bash

```
sudo apt-get install -y maven
```

Add Azure service principal to a Jenkins credential

The following steps show how to specify your Azure credential:

1. Make sure the Credentials plug-in is installed.
2. Within the Jenkins dashboard, select **Credentials** -> **System** ->.
3. Select **Global credentials(unrestricted)**.
4. Select **Add Credentials** to add a Microsoft Azure service principal. Make sure that the credential kind is ***Username with password*** and enter the following items:
   - **Username**: Service principal `appId`
   - **Password**: Service principal `password`
   - **ID**: Credential identifier (such as `AzureServicePrincipal`)

Create an Azure App Service for deploying the Java web app

Use az appservice plan create to create an Azure App Service plan with the **FREE** pricing tier:

Azure CLI

```
az appservice plan create \
    --name <app_service_plan> \
```

```
    --resource-group <resource_group> \
    --sku FREE
```

Key points:

- The appservice plan defines the physical resources used to host your apps.
- All applications assigned to an appservice plan share these resources.
- Appservice plans allow you to save cost when hosting multiple apps.

Create an Azure web app

Use az webapp create to create a web app definition in the `myAppServicePlan` App Service plan.

Azure CLI

```
az webapp create \
    --name <app_name> \
    --resource-group <resource_group> \
    --plan <app_service_plan>
```

Key points:

- The web app definition provides a URL to access your application with and configures several options to deploy your code to Azure.
- Substitute the `<app_name>` placeholder with a unique app name.
- The app name is part of the default domain name for the web app. Therefore, the name needs to be unique across all apps in Azure.
- You can map a custom domain name entry to the web app before you expose it to your users.

Configure Java

Use az appservice web config update to set up the Java runtime configuration for the app:

Azure CLI

```
az webapp config set \
    --name <app_name> \
    --resource-group <resource_group> \
    --java-version 1.8 \
    --java-container Tomcat \
    --java-container-version 8.0
```

Prepare a GitHub repository

1. Open the Simple Java Web App for Azure repo.
2. Select the **Fork** button to fork the repo to your own GitHub account.
3. Open the **Jenkinsfile** file by clicking on the file name.
4. Select the pencil icon to edit the file.
5. Update the subscription ID and tenant ID.

```groovy
  withEnv(['AZURE_SUBSCRIPTION_ID=<subscription_id>',
        'AZURE_TENANT_ID=<tenant_id>'])
```

6. Update the resource group and name of your web app on line 22 and 23 respectively.

```groovy
def resourceGroup = '<resource_group>'
def webAppName = '<app_name>'
```

7. Update the credential ID in your Jenkins instance

```groovy
withCredentials([usernamePassword(credentialsId: '<service_princial>',
passwordVariable: 'AZURE_CLIENT_SECRET', usernameVariable:
'AZURE_CLIENT_ID')]) {
```

Create Jenkins pipeline

Do the following to create a Jenkins pipeline:

1. Open Jenkins in a web browser.
2. Select **New Item**.
3. Enter a name for the job.

4. Select **Pipeline**.
5. Select **OK**.
6. Select **Pipeline**.
7. For **Definition**, select **Pipeline script from SCM**.
8. For **SCM**, select **Git**.
9. Enter the GitHub URL for your forked repo: `https:\<forked_repo\>.git`
10. Select **Save**

Test your pipeline

1. Go to the pipeline you created
2. Select **Build Now**
3. After the build completes, select **Console Output** to see build details.

Verify your web app

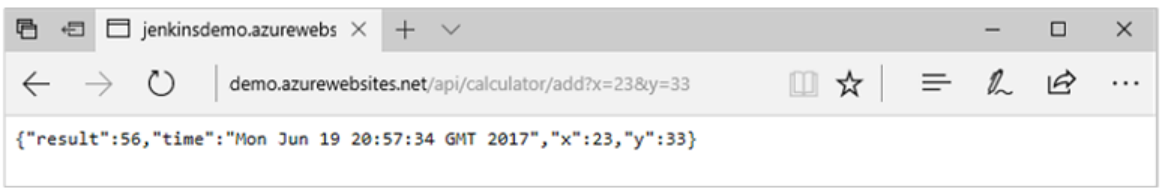
Do the following to verify the WAR file is deployed successfully to your web app:

1. Browse to the following URL:
   `http://<app_name>.azurewebsites.net/api/calculator/ping`
2. You should see text similar to the following:

   Output

   ```
   Welcome to Java Web App!!! This is updated!
   Today's date
   ```

3. Browse to the following URL (substitute <x> and <y> with two values to be summed): http://<app_name>.azurewebsites.net/api/calculator/add?x=<x>&y=<y>.



Deploy to Azure App Service on Linux

App Service can also host web apps natively on Linux for supported application stacks. It can also run custom Linux containers (also known as Web App for Containers.)

You can modify the script to deploy to an Azure App Service on Linux. App Service on Linux supports Docker. As such, you provide a Dockerfile that packages your web app with service runtime into a Docker image. The plug-in builds the image, pushes it to a Docker registry, and deploys the image to your web app.

1. Refer to Migrate custom software to Azure App Service using a custom container to create an Azure App Service on Linux and an Azure Container Registry.

   Azure CLI

   ```
       az group create --name myResourceGroup2 --location westus2
       az acr create --name myACRName --resource-group myResourceGroup2 --
   sku Basic --admin-enabled true
       az appservice plan create --name myAppServicePlan --resource-group
   myResourceGroup2 --is-linux
       az webapp create --resource-group myResourceGroup2 --plan
   myAppServicePlan --name myApp --deployment-container-image-name
   myACRName.azurecr.io/calculator:latest
   ```

2. Install Docker on your Jenkins.

3. Make sure Docker Pipeline plug-in is installed.

4. In the same Simple Java Web App for Azure repo you forked, edit the **Jenkinsfile2** file as follows:

   a. Update the subscription ID and tenant ID.

   groovy

   ```
    withEnv(['AZURE_SUBSCRIPTION_ID=<mySubscriptionId>',
           'AZURE_TENANT_ID=<myTenantId>']) {
   ```

   b. Update to the names of your resource group, web app, and ACR (replacing the placeholders with your values).

   Bash

   ```
   def webAppResourceGroup = '<resource_group>'
   def webAppName = '<app_name>'
   def acrName = '<registry>'
   ```

c. Update `<azsrvprincipal\>` to your credential ID

```
Bash

withCredentials([usernamePassword(credentialsId:
'<service_principal>', passwordVariable: 'AZURE_CLIENT_SECRET',
usernameVariable: 'AZURE_CLIENT_ID')]) {
```

5. Create a new Jenkins pipeline as you did when deploying to Azure web app in Windows using `Jenkinsfile2`.

6. Run your new job.

7. To verify, in Azure CLI, run the following command:

```
Azure CLI

az acr repository list -n <myRegistry> -o json
```

You should see results similar to the following:

```
Output

[
"calculator"
]
```

8. Browse to `http://<app_name>.azurewebsites.net/api/calculator/ping` (replacing the placeholder). You should see similar results to the following:

```
Output

Welcome to Java Web App!!! This is updated!
Today's date
```

9. Browse to `http://<app_name>.azurewebsites.net/api/calculator/add?x=<x>&y=<y>` (replacing the placeholders). The values you specify for `x` and `y` are summed and displayed.

Next steps

Jenkins on Azure

Tutorial: Deploy to Linux virtual machine using Jenkins and Azure DevOps Services

Article • 05/30/2023

Continuous integration (CI) and continuous deployment (CD) form a pipeline by which you can build, release, and deploy your code. Azure DevOps Services provides a complete, fully featured set of CI/CD automation tools for deployment to Azure. Jenkins is a popular third-party CI/CD server-based tool that also provides CI/CD automation. You can use Azure DevOps Services and Jenkins together to customize how you deliver your cloud app or service.

In this tutorial, you use Jenkins to build a Node.js web app. You then use Azure DevOps to deploy it

to a deployment group that contains Linux virtual machines (VMs). You learn how to:

- ✓ Get the sample app.
- ✓ Configure Jenkins plug-ins.
- ✓ Configure a Jenkins Freestyle project for Node.js.
- ✓ Configure Jenkins for Azure DevOps Services integration.
- ✓ Create a Jenkins service endpoint.
- ✓ Create a deployment group for the Azure virtual machines.
- ✓ Create an Azure Pipelines release pipeline.
- ✓ Execute manual and CI-triggered deployments.

Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free Azure account before you begin.
- **Jenkins server**: If you don't have a Jenkins server installed, create a Jenkins server on Azure.

> ⓘ **Note**
>
> For more information, see **Connect to Azure DevOps Services**.

- You need a Linux virtual machine for a deployment target. For more information, see Create and manage Linux VMs with the Azure CLI.
- Open inbound port 80 for your virtual machine. For more information, see Create network security groups using the Azure portal.

Get the sample app

You need an app to deploy, stored in a Git repository. For this tutorial, we recommend that you use this sample app available from GitHub. This tutorial contains a sample script that's used for installing Node.js and an application. If you want to work with your own repository, you should configure a similar sample.

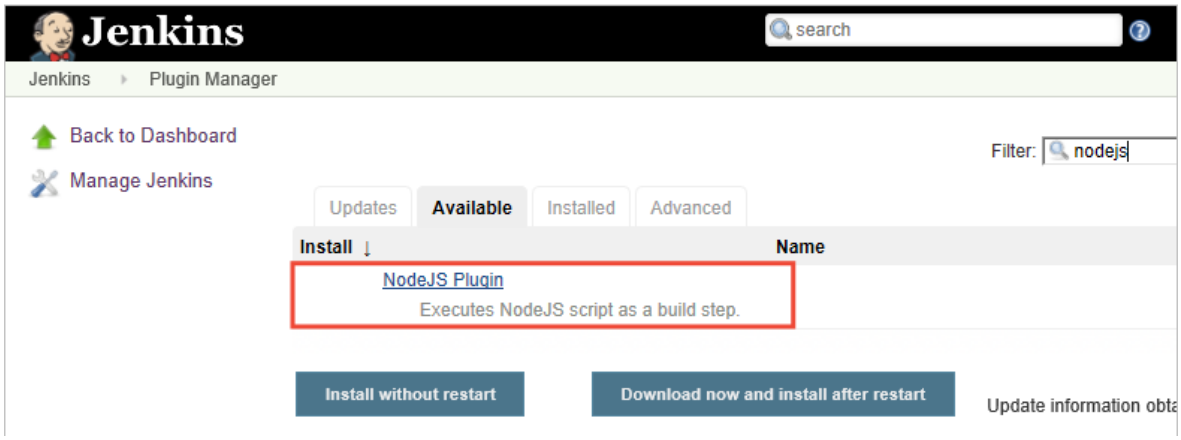
Create a fork of this app and take note of the location (URL) for use in later steps of this tutorial. For more information, see Fork a repo.

> **Note**
>
> The app was built through **Yeoman**. It uses Express, bower, and grunt. And it has some npm packages as dependencies. The sample also contains a script that sets up Nginx and deploys the app. It is executed on the virtual machines. Specifically, the script:
>
> 1. Installs Node, Nginx, and PM2.
> 2. Configures Nginx and PM2.
> 3. Starts the Node app.

Configure Jenkins plug-ins

First, you must configure two Jenkins plug-ins: **NodeJS** and **VS Team Services Continuous Deployment**.

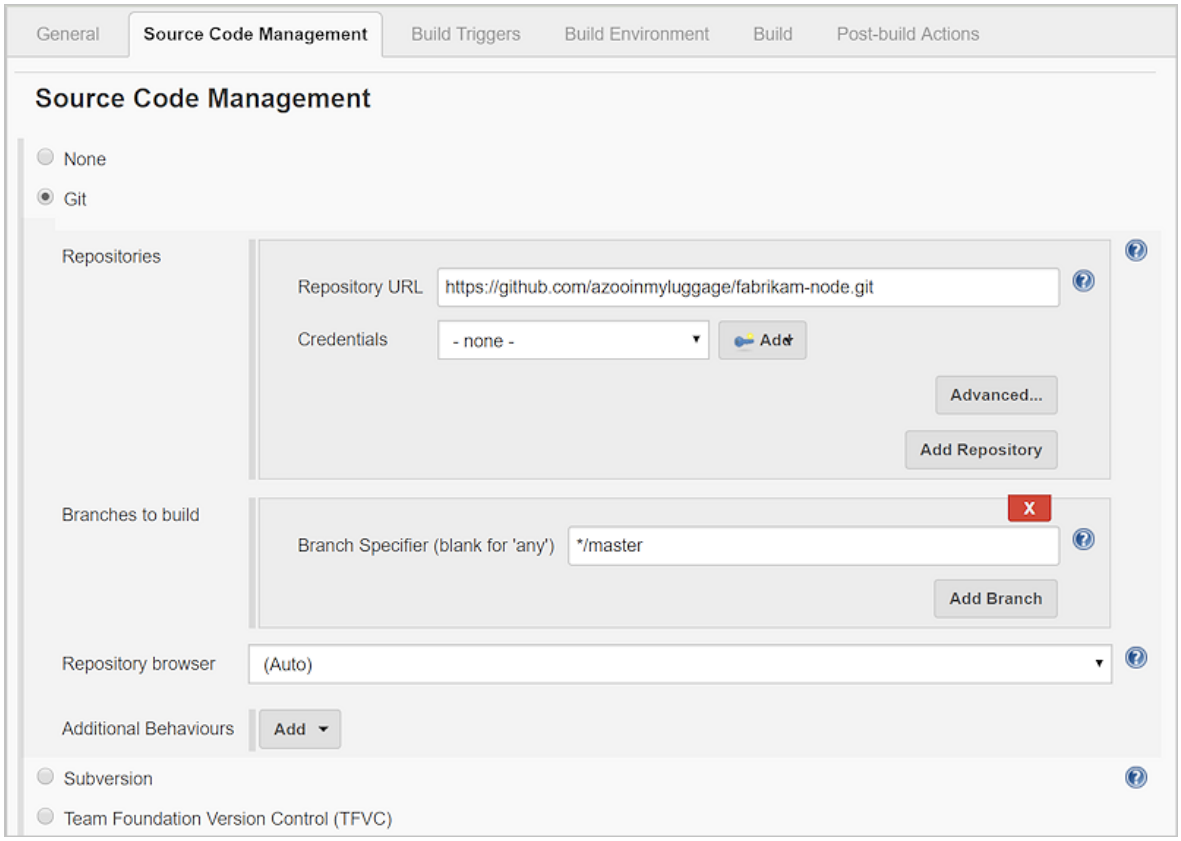
1. Open your Jenkins account and select **Manage Jenkins**.
2. On the **Manage Jenkins** page, select **Manage Plugins**.
3. Filter the list to locate the **NodeJS** plug-in, and select the **Install without restart** option.



4. Filter the list to find the **VS Team Services Continuous Deployment** plug-in and select the **Install without restart** option.
5. Go back to the Jenkins dashboard and select **Manage Jenkins**.
6. Select **Global Tool Configuration**. Find **NodeJS** and select **NodeJS installations**.
7. Select the **Install automatically** option, and then enter a **Name** value.
8. Select **Save**.

Configure a Jenkins Freestyle project for Node.js

1. Select **New Item**. Enter an item name.
2. Select **Freestyle project**. Select **OK**.
3. On the **Source Code Management** tab, select **Git** and enter the details of the repository and the branch that contain your app code.



4. On the **Build Triggers** tab, select **Poll SCM** and enter the schedule `H/03 * * * *` to poll the Git repository for changes every three minutes.
5. On the **Build Environment** tab, select **Provide Node & npm bin/ folder PATH** and select the **NodeJS Installation** value. Leave **npmrc file** set to **use system default**.
6. On the **Build** tab, select **Execute shell** and enter the command `npm install` to ensure that all dependencies are updated.

Configure Jenkins for Azure DevOps Services integration

> **Note**
>
> Ensure that the personal access token (PAT) you use for the following steps contains the *Release* (read, write, execute and manage) permission in Azure DevOps Services.

1. Create a PAT in your Azure DevOps Services organization if you don't already have one. Jenkins requires this information to access your Azure DevOps Services organization. Be sure to store the token information for upcoming steps in this section.

   To learn how to generate a token, read How do I create a personal access token for Azure DevOps Services?.

2. In the **Post-build Actions** tab, select **Add post-build action**. Select **Archive the artifacts**.

3. For **Files to archive**, enter `**/*` to include all files.

4. To create another action, select **Add post-build action**.

5. Select **Trigger release in TFS/Team Services**. Enter the URI for your Azure DevOps Services organization, such as **https://{your-organization-name}.visualstudio.com**.

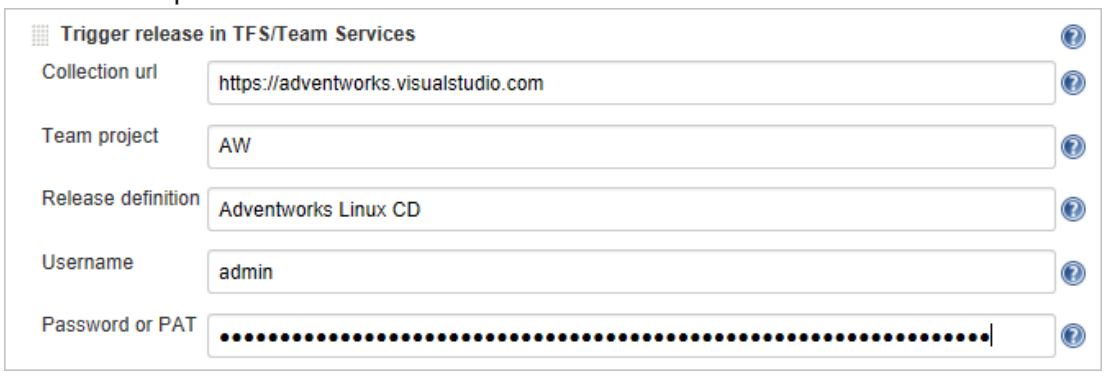
6. Enter the **Project** name.

7. Choose a name for the release pipeline. (You create this release pipeline later in Azure DevOps Services.)

8. Choose credentials to connect to your Azure DevOps Services or Azure DevOps Server environment:

   - Leave **Username** blank if you are using Azure DevOps Services.

- Enter a username and password if you are using an on-premises version of Azure DevOps Server.

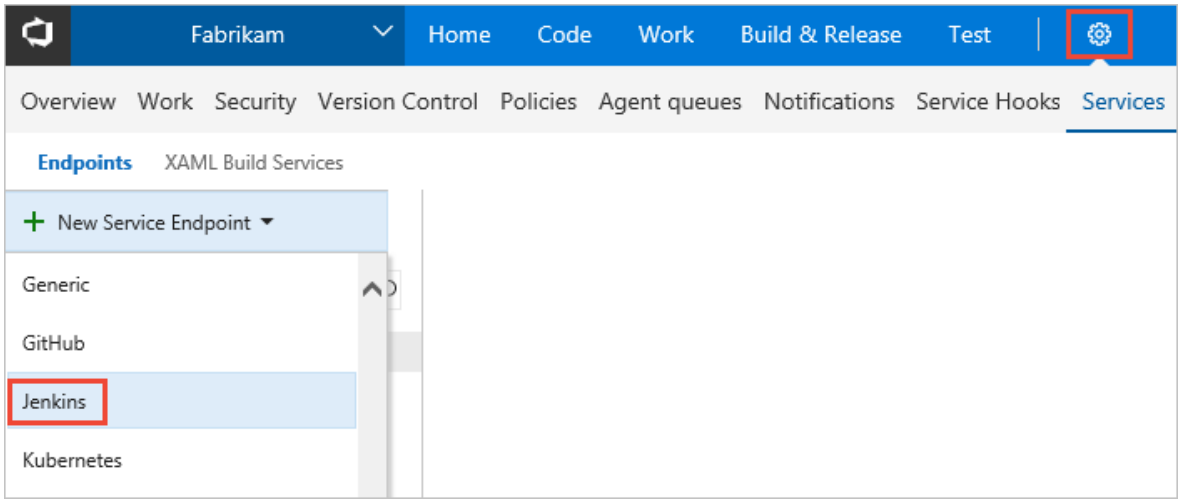


9. Save the Jenkins project.

Create a Jenkins service endpoint

A service endpoint allows Azure DevOps Services to connect to Jenkins.

1. Open the **Services** page in Azure DevOps Services, open the **New Service Endpoint** list, and select **Jenkins**.



2. Enter a name for the connection.
3. Enter the URL of your Jenkins server, and select the **Accept untrusted SSL certificates** option. An example URL is **http://{YourJenkinsURL}.westcentralus.cloudapp.azure.com**.
4. Enter the username and password for your Jenkins account.
5. Select **Verify connection** to check that the information is correct.
6. Select **OK** to create the service endpoint.

Create a deployment group for Azure virtual machines

You need a deployment group to register the Azure DevOps Services agent so the release pipeline can be deployed to your virtual machine. Deployment groups make it easy to define logical groups of target machines for deployment, and to install the required agent on each machine.

> **Note**
>
> In the following procedure, be sure to install the prerequisites and *don't run the script with sudo privileges.*

1. Open the **Releases** tab of the **Build & Release** hub, open **Deployment groups**, and select **+ New**.
2. Enter a name for the deployment group, and an optional description. Then select **Create**.
3. Choose the operating system for your deployment target virtual machine. For example, select **Ubuntu 16.04+**.
4. Select **Use a personal access token in the script for authentication**.
5. Select the **System prerequisites** link. Install the prerequisites for your operating system.
6. Select **Copy script to clipboard** to copy the script.
7. Log in to your deployment target virtual machine and run the script. Don't run the script with sudo privileges.
8. After the installation, you are prompted for deployment group tags. Accept the defaults.
9. In Azure DevOps Services, check for your newly registered virtual machine in **Targets** under **Deployment Groups**.

Create an Azure Pipelines release pipeline

A release pipeline specifies the process that Azure Pipelines uses to deploy the app. In this example, you execute a shell script.

To create the release pipeline in Azure Pipelines:

1. Open the **Releases** tab of the **Build & Release** hub, and select **Create release pipeline**.
2. Select the **Empty** template by choosing to start with an **Empty process**.
3. In the **Artifacts** section, select **+ Add Artifact** and choose **Jenkins** for **Source type**. Select your Jenkins service endpoint connection. Then select the Jenkins source job and select **Add**.
4. Select the ellipsis next to **Environment 1**. Select **Add deployment group phase**.

5. Choose your deployment group.
6. Select + to add a task to **Deployment group phase**.
7. Select the **Shell Script** task and select **Add**. The **Shell Script** task provides the configuration for a script to run on each server in order to install Node.js and start the app.
8. For **Script Path**, enter **$(System.DefaultWorkingDirectory)/Fabrikam-Node/deployscript.sh**.
9. Select **Advanced**, and then enable **Specify Working Directory**.
10. For **Working Directory**, enter **$(System.DefaultWorkingDirectory)/Fabrikam-Node**.
11. Edit the name of the release pipeline to the name that you specified on the **Post-build Actions** tab of the build in Jenkins. Jenkins requires this name to be able to trigger a new release when the source artifacts are updated.
12. Select **Save** and select **OK** to save the release pipeline.

Execute manual and CI-triggered deployments

1. Select + **Release** and select **Create Release**.
2. Select the build that you completed in the highlighted drop-down list, and select **Queue**.
3. Choose the release link in the pop-up message. For example: "Release **Release-1** has been created."
4. Open the **Logs** tab to watch the release console output.
5. In your browser, open the URL of one of the servers that you added to your deployment group. For example, enter **http://{your-server-ip-address}**.
6. Go to the source Git repository and modify the contents of the **h1** heading in the file app/views/index.jade with some changed text.
7. Commit your change.
8. After a few minutes, you will see a new release created on the **Releases** page of Azure DevOps. Open the release to see the deployment taking place. Congratulations!

Troubleshooting the Jenkins plug-in

If you encounter any bugs with the Jenkins plug-ins, file an issue in the Jenkins JIRA for the specific component.

Next steps

In this tutorial, you automated the deployment of an app to Azure by using Jenkins for build and Azure DevOps Services for release. You learned how to:

- ✓ Build your app in Jenkins.
- ✓ Configure Jenkins for Azure DevOps Services integration.
- ✓ Create a deployment group for the Azure virtual machines.
- ✓ Create an Azure Pipeline that configures the VMs and deploys the app.

To learn about how to use Azure Pipelines for both Build and Release steps, refer to this.

To learn about how to author a YAML based CI/CD pipeline to deploy to VMs, advance to the next tutorial.

Jenkins on Azure

Tutorial: Deploy to Azure Functions using Jenkins

Article • 05/30/2023

> **Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

Azure Functions is a serverless compute service. Using Azure Functions, you can run code on-demand without provisioning or managing infrastructure. This tutorial shows how to deploy a Java function to Azure Functions using the Azure Functions plug-in.

Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free account before you begin.
- **Jenkins server**: If you don't have a Jenkins server installed, refer to the article, Create a Jenkins server on Azure.

View the source code

The source code used for this tutorial is located in the Visual Studio China GitHub repo.

Create a Java function

To create a Java function with the Java runtime stack, use either the Azure portal or the Azure CLI.

The following steps show how to create a Java function using the Azure CLI:

1. Create a resource group, replacing the <**resource_group**> placeholder with your resource group name.

   Azure CLI

```
az group create --name <resource_group> --location eastus
```

2. Create an Azure storage account, replacing the placeholders with the appropriate values.

Azure CLI

```
az storage account create --name <storage_account> --location eastus --
resource-group <resource_group> --sku Standard_LRS
```

3. Create the test function app, replacing the placeholders with the appropriate values.

Azure CLI

```
az functionapp create --resource-group <resource_group> --runtime java
--consumption-plan-location eastus --name <function_app> --storage-
account <storage_account> --functions-version 2
```

Prepare Jenkins server

The following steps explain how to prepare the Jenkins server:

1. Deploy a Jenkins server on Azure. If you don't already have an instance of the Jenkins server installed, the article, Create a Jenkins server on Azure guides you through the process.

2. Sign in to the Jenkins instance with SSH.

3. On the Jenkins instance, install Az CLI, version 2.0.67 or higher.

4. Install maven using the following command:

Bash

```
sudo apt install -y maven
```

5. On the Jenkins instance, install the Azure Functions Core Tools by issuing the following commands at a terminal prompt:

Bash

```
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor
> microsoft.gpg
```

```
sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg
sudo sh -c 'echo "deb [arch=amd64]
https://packages.microsoft.com/repos/microsoft-ubuntu-$(lsb_release -
cs)-prod $(lsb_release -cs) main" >
/etc/apt/sources.list.d/dotnetdev.list'
cat /etc/apt/sources.list.d/dotnetdev.list
sudo apt-get update
sudo apt-get install azure-functions-core-tools-3
```

6. Jenkins needs an Azure service principal to authenticate and access Azure resources. Refer to the Deploy to Azure App Service for step-by-step instructions.

7. Make sure the Credentials plug-in is installed.

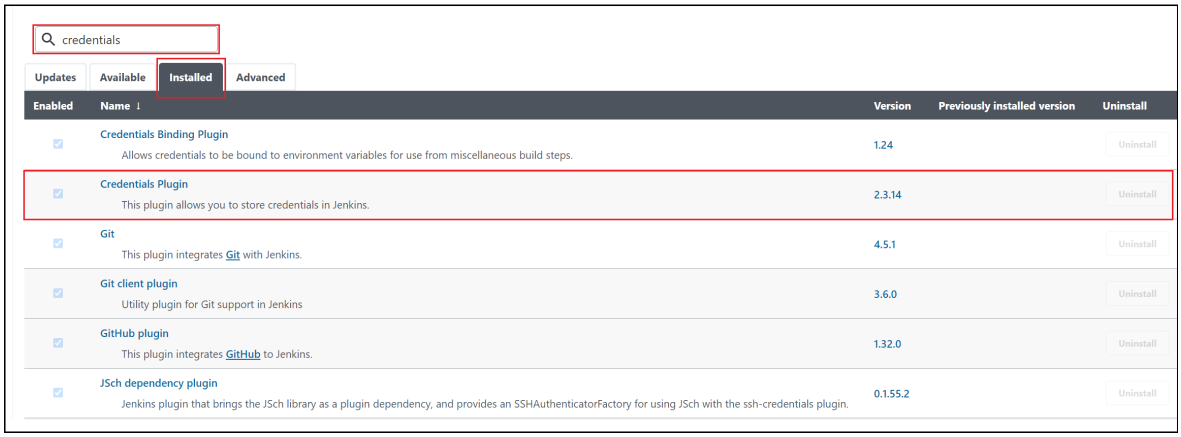
   a. From the menu, select **Manage Jenkins**.

   b. Under **System Configuration**, select **Manage plug-in**.

   c. Select the **Installed** tab.

   d. In the **filter** field, enter `credentials`.

   e. Verify that the **Credentials plug-in** is installed. If not, you'll need to install it from the **Available** tab.



8. From the menu, select **Manage Jenkins**.

9. Under **Security**, select **Manage Credentials**.

10. Under **Credentials**, select **(global)**.

11. From the menu, select **Add Credentials**.

12. Enter the following values for your Microsoft Azure service principal:

    - **Kind**: Select the value: ***Username with password***.
    - **Username**: Specify the `appId` of the service principal created.

- **Password**: Specify the `password` (secret) of the service principal.
- **ID**: Specify the credential identifier, such as `azuresp`.

13. Select **OK**.

Fork the sample GitHub repo

1. Sign in to the GitHub repo for the odd or even sample app.
2. In the upper-right corner in GitHub, choose **Fork**.
3. Follow the prompts to select your GitHub account and finish forking.

Create a Jenkins Pipeline

In this section, you create the Jenkins Pipeline.

1. In the Jenkins dashboard, create a Pipeline.
2. Enable **Prepare an environment for the run**.
3. In the **Pipeline->Definition** section, select **Pipeline script from SCM**.
4. Enter your GitHub fork's URL and script path ("doc/resources/jenkins/JenkinsFile") to use in the JenkinsFile example.

Node.js

```
node {
withEnv(['AZURE_SUBSCRIPTION_ID=99999999-9999-9999-9999-999999999999',
        'AZURE_TENANT_ID=99999999-9999-9999-9999-999999999999']) {
    stage('Init') {
        cleanWs()
        checkout scm
    }

    stage('Build') {
        sh 'mvn clean package'
    }

    stage('Publish') {
        def RESOURCE_GROUP = '<resource_group>'
        def FUNC_NAME = '<function_app>'
        // login Azure
        withCredentials([usernamePassword(credentialsId: 'azuresp',
passwordVariable: 'AZURE_CLIENT_SECRET', usernameVariable:
'AZURE_CLIENT_ID')]) {
        sh '''
```

```
            az login --service-principal -u $AZURE_CLIENT_ID -p
$AZURE_CLIENT_SECRET -t $AZURE_TENANT_ID
            az account set -s $AZURE_SUBSCRIPTION_ID
        '''
        }
        sh 'cd $PWD/target/azure-functions/odd-or-even-function-sample
&& zip -r ../../../archive.zip ./* && cd -'
        sh "az functionapp deployment source config-zip -g
$RESOURCE_GROUP -n $FUNC_NAME --src archive.zip"
        sh 'az logout'
        }
    }
 }
```

Build and deploy

It's now time to run the Jenkins job.

1. First, obtain the authorization key via the instructions in the Azure Functions HTTP triggers and bindings article.

2. In your browser, enter the app's URL. Replace the placeholders with the appropriate values and specify a numeric value for **<input_number>** as input for the Java function.

```
https://<function_app>.azurewebsites.net/api/HttpTrigger-Java?code=
<authorization_key>&number=<input_number>
```

3. You'll see results similar to the following example output (where an odd number - 365 - was used as a test):

Output

```
The number 365 is Odd.
```

Clean up resources

If you're not going to continue to use this application, delete the resources you created with the following step:

Azure CLI

```
az group delete -y --no-wait -n <resource_group>
```

Next steps

Azure Functions

Tutorial: Use Azure Storage for build artifacts

Article • 08/08/2021

> **Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

This article illustrates how to use Blob storage as a repository of build artifacts created by a Jenkins continuous integration (CI) solution, or as a source of downloadable files to be used in a build process. One of the scenarios where you would find this solution useful is when you're coding in an agile development environment (using Java or other languages), builds are running based on continuous integration, and you need a repository for your build artifacts, so that you could, for example, share them with other organization members, your customers, or maintain an archive. Another scenario is when your build job itself requires other files, for example, dependencies to download as part of the build input.

Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free Azure account before you begin.
- **Jenkins server**: If you don't have a Jenkins server installed, create a Jenkins server on Azure.
- **Azure CLI**: Install Azure CLI (version 2.0.67 or higher) on the Jenkins server.
- **Azure storage account**: If you don't already have a storage account, create a Storage Account.

Add Azure credential needed to execute Azure CLI

1. Browse to the Jenkins portal.
2. From the menu, select **Manage Jenkins**.

3. Select **Manage Credentials**.

4. Select the **global** domain.

5. Select **Add Credentials**.

6. Fill out the required fields as follows:

    - **Kind**: Select **Username with password**.
    - **Username**: Specify the `appId` of the service principal.
    - **Password**: Specify the `password` of the service principal.
    - **ID**: Specify a credential identifer, such as `azuresp`.
    - **Description**: Optionally, include a meaningful description for your environment.

7. Select **OK** to create the credential.

Create a pipeline job to upload build artifacts

The following steps guide you through creating a pipeline job. The pipeline job creates several files and uploads the files to your storage account using Azure CLI.

1. From the Jenkins dashboard, select **New Item**.

2. Name the job **myjob**, select **Pipeline**, and then select **OK**.

3. In the **Pipeline** section of the job configuration, select **Pipeline script** and paste the following into **Script**. Edit the placeholders to match the values for your environment.

    groovy

    ```
    pipeline {
      agent any
      environment {
        AZURE_SUBSCRIPTION_ID='99999999-9999-9999-9999-999999999999'
        AZURE_TENANT_ID='99999999-9999-9999-9999-999999999999'
        AZURE_STORAGE_ACCOUNT='myStorageAccount'
      }
      stages {
        stage('Build') {
          steps {
            sh 'rm -rf *'
            sh 'mkdir text'
            sh 'echo Hello Azure Storage from Jenkins > ./text/hello.txt'
            sh 'date > ./text/date.txt'
          }
    ```

```
      post {
        success {
          withCredentials([usernamePassword(credentialsId: 'azuresp',
                          passwordVariable: 'AZURE_CLIENT_SECRET',
                          usernameVariable: 'AZURE_CLIENT_ID')]) {
            sh '''
              echo $container_name
              # Login to Azure with ServicePrincipal
              az login --service-principal -u $AZURE_CLIENT_ID -p
$AZURE_CLIENT_SECRET -t $AZURE_TENANT_ID
              # Set default subscription
              az account set --subscription $AZURE_SUBSCRIPTION_ID
              # Execute upload to Azure
              az storage container create --account-name
$AZURE_STORAGE_ACCOUNT --name $JOB_NAME --auth-mode login
              az storage blob upload-batch --destination ${JOB_NAME} --
source ./text --account-name $AZURE_STORAGE_ACCOUNT
              # Logout from Azure
              az logout
            '''
          }
        }
      }
    }
  }
}
```

4. Select **Build Now** to run **myjob**.

5. Examine the console output for status. When the post-build action uploads the build artifacts, status messages for Azure storage are written to the console.

6. If you encounter an error similar to the following, it means that you need to grant access at the container level: `ValidationError: You do not have the required permissions needed to perform this operation.` If you receive this error message, refer to the following articles to resolve:

   - Choose how to authorize access to blob data with Azure CLI - Azure Storage
   - Use the Azure portal to assign an Azure role for data access - Azure Storage

7. Upon successful completion of the job, examine the build artifacts by opening the public blob:
   a. Sign in to the Azure portal.
   b. Select **Storage**.
   c. Select the storage account name that you used for Jenkins.
   d. Select **Containers**.
   e. Select the container named **myjob**, within the list of blobs.
   f. You should see the following two files: **hello.txt** and **date.txt**.

g. Copy the URL for either of these items and paste it in your browser.
h. You see the text file that was uploaded as a build artifact.

Key points:

- Container names and blob names are lowercase (and case-sensitive) in Azure storage.

Create a pipeline job to download from Azure Blob Storage

The following steps show how to configure a pipeline job to download items from Azure Blob Storage.

1. In the **Pipeline** section of the job configuration, select **Pipeline script** and paste the following in **Script**. Edit the placeholders to match the values for your environment.

```groovy
pipeline {
  agent any
  environment {
    AZURE_SUBSCRIPTION_ID='99999999-9999-9999-9999-999999999999'
    AZURE_TENANT_ID='99999999-9999-9999-9999-999999999999'
    AZURE_STORAGE_ACCOUNT='myStorageAccount'
  }
  stages {
    stage('Build') {
      steps {
        withCredentials([usernamePassword(credentialsId: 'azuresp',
                        passwordVariable: 'AZURE_CLIENT_SECRET',
                        usernameVariable: 'AZURE_CLIENT_ID')]) {
          sh '''
            # Login to Azure with ServicePrincipal
            az login --service-principal -u $AZURE_CLIENT_ID -p
$AZURE_CLIENT_SECRET -t $AZURE_TENANT_ID
            # Set default subscription
            az account set --subscription $AZURE_SUBSCRIPTION_ID
            # Execute upload to Azure
            az storage blob download --account-name
$AZURE_STORAGE_ACCOUNT --container-name myjob --name hello.txt --file
${WORKSPACE}/hello.txt --auth-mode login
            # Logout from Azure
            az logout
          '''
        }
      }
    }
```

```
    }
}
```

2. After running a build, check the build history console output. Alternatively, you can also look at your download location to see if the blobs you expected were successfully downloaded.

Next steps

Jenkins on Azure

Tutorial: Create a Jenkins pipeline using GitHub and Docker

Article • 01/22/2021

> **ⓘ Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

To automate the build and test phase of application development, you can use a continuous integration and deployment (CI/CD) pipeline. In this tutorial, you create a CI/CD pipeline on an Azure VM including how to:

- ✓ Create a Jenkins VM
- ✓ Install and configure Jenkins
- ✓ Create webhook integration between GitHub and Jenkins
- ✓ Create and trigger Jenkins build jobs from GitHub commits
- ✓ Create a Docker image for your app
- ✓ Verify GitHub commits build new Docker image and updates running app

This tutorial uses the CLI within the Azure Cloud Shell, which is constantly updated to the latest version. To open the Cloud Shell, select **Try it** from the top of any code block.

If you choose to install and use the CLI locally, this tutorial requires that you are running the Azure CLI version 2.0.30 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create Jenkins instance

In a previous tutorial on How to customize a Linux virtual machine on first boot, you learned how to automate VM customization with cloud-init. This tutorial uses a cloud-init file to install Jenkins and Docker on a VM. Jenkins is a popular open-source automation server that integrates seamlessly with Azure to enable continuous integration (CI) and continuous delivery (CD). For more tutorials on how to use Jenkins, see the Jenkins in Azure hub.

In your current shell, create a file named *cloud-init-jenkins.txt* and paste the following configuration. For example, create the file in the Cloud Shell not on your local machine. Enter `sensible-editor cloud-init-jenkins.txt` to create the file and see a list of available editors. Make sure that the whole cloud-init file is copied correctly, especially the first line:

YAML

```
#cloud-config
package_upgrade: true
write_files:
  - path: /etc/systemd/system/docker.service.d/docker.conf
    content: |
      [Service]
        ExecStart=
        ExecStart=/usr/bin/dockerd
  - path: /etc/docker/daemon.json
    content: |
      {
        "hosts": ["fd://","tcp://127.0.0.1:2375"]
      }
runcmd:
  - apt install openjdk-8-jre-headless -y
  - wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo
apt-key add -
  - sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
  - apt-get update && apt-get install jenkins -y
  - curl -sSL https://get.docker.com/ | sh
  - usermod -aG docker azureuser
  - usermod -aG docker jenkins
  - service jenkins restart
```

Before you can create a VM, create a resource group with az group create. The following example creates a resource group named *myResourceGroupJenkins* in the *eastus* location:

Azure CLI

```
az group create --name myResourceGroupJenkins --location eastus
```

Now create a VM with az vm create. Use the `--custom-data` parameter to pass in your cloud-init config file. Provide the full path to *cloud-init-jenkins.txt* if you saved the file outside of your present working directory.

Azure CLI

```
az vm create --resource-group myResourceGroupJenkins \
    --name myVM \
    --image UbuntuLTS \
    --admin-username azureuser \
    --generate-ssh-keys \
    --custom-data cloud-init-jenkins.txt
```

It takes a few minutes for the VM to be created and configured.

To allow web traffic to reach your VM, use az vm open-port to open port *8080* for Jenkins traffic and port *1337* for the Node.js app that is used to run a sample app:

Azure CLI

```
az vm open-port --resource-group myResourceGroupJenkins --name myVM --port
8080 --priority 1001
az vm open-port --resource-group myResourceGroupJenkins --name myVM --port
1337 --priority 1002
```

Configure Jenkins

To access your Jenkins instance, obtain the public IP address of your VM:

Azure CLI

```
az vm show --resource-group myResourceGroupJenkins --name myVM -d --query
[publicIps] --o tsv
```

For security purposes, you need to enter the initial admin password that is stored in a text file on your VM to start the Jenkins install. Use the public IP address obtained in the previous step to SSH to your VM:

Bash

```
ssh azureuser@<publicIps>
```

Verify Jenkins is running using the `service` command:

Bash

```
$ service jenkins status
● jenkins.service - LSB: Start Jenkins at boot time
   Loaded: loaded (/etc/init.d/jenkins; generated)
   Active: active (exited) since Tue 2019-02-12 16:16:11 UTC; 55s ago
     Docs: man:systemd-sysv-generator(8)
```

```
     Tasks: 0 (limit: 4103)
    CGroup: /system.slice/jenkins.service

Feb 12 16:16:10 myVM systemd[1]: Starting LSB: Start Jenkins at boot time...
...
```

View the `initialAdminPassword` for your Jenkins install and copy it:

```Bash
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

If the file isn't available yet, wait a couple more minutes for cloud-init to complete the Jenkins and Docker install.

Now open a web browser and go to `http://<publicIps>:8080`. Complete the initial Jenkins setup as follows:

- Choose **Select plug-ins to install**
- Search for *GitHub* in the text box across the top. Check the box for *GitHub*, then select **Install**
- Create the first admin user. Enter a username, such as **admin**, then provide your own secure password. Finally, type a full name and e-mail address.
- Select **Save and Finish**
- Once Jenkins is ready, select **Start using Jenkins**
  - If your web browser displays a blank page when you start using Jenkins, restart the Jenkins service. From your SSH session, type `sudo service jenkins restart`, then refresh you web browser.
- If needed, log in to Jenkins with the username and password you created.

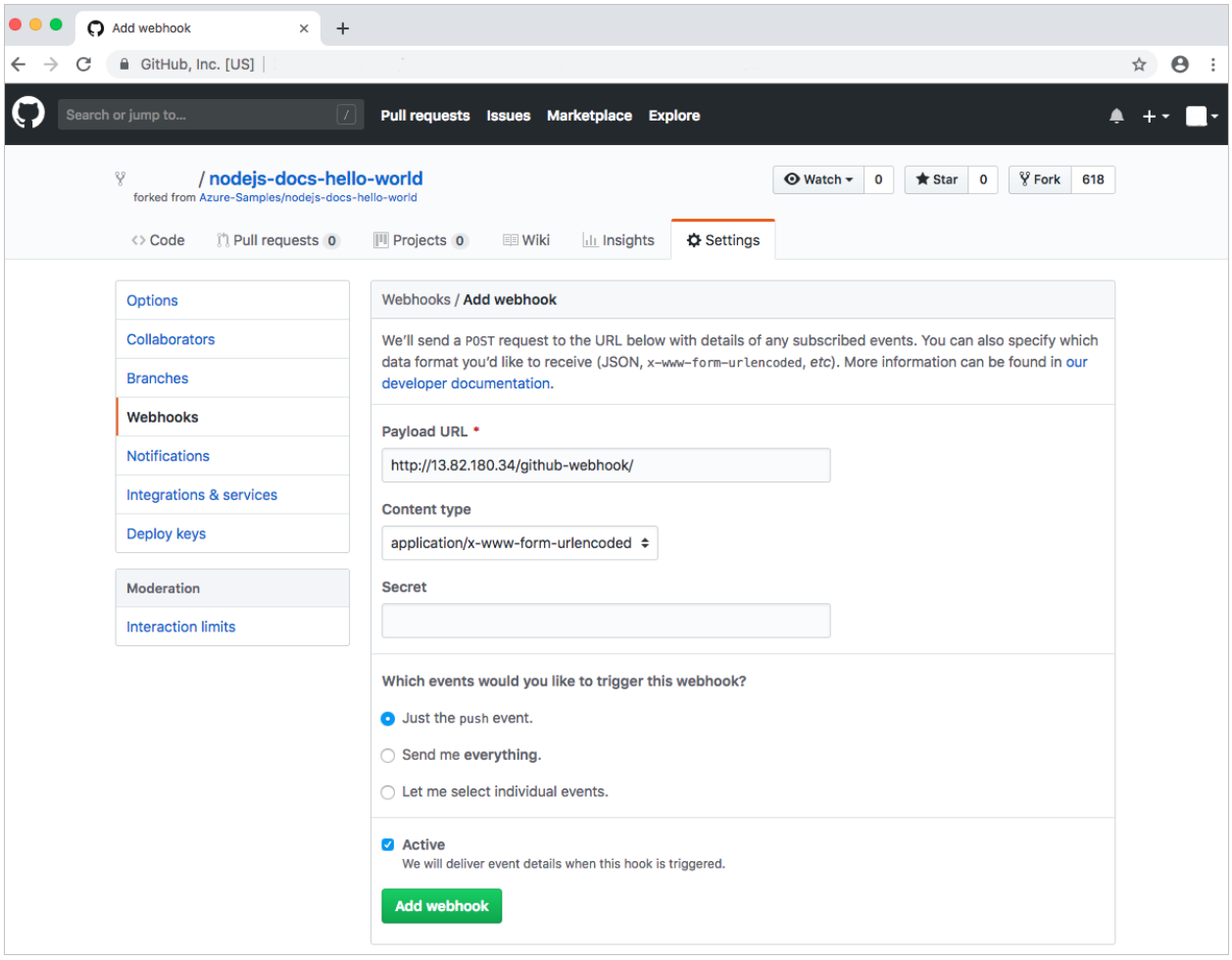
Create GitHub webhook

To configure the integration with GitHub, open the Node.js Hello World sample app from the Azure samples repo. To fork the repo to your own GitHub account, select the **Fork** button in the top right-hand corner.

Create a webhook inside the fork you created:

- Select **Settings**, then select **Webhooks** on the left-hand side.
- Choose **Add webhook**, then enter *Jenkins* in filter box.
- For the **Payload URL**, enter `http://<publicIps>:8080/github-webhook/`. Make sure you include the trailing /
- For **Content type**, select *application/x-www-form-urlencoded*.

- For **Which events would you like to trigger this webhook?**, select *Just the push event.*
- Set **Active** to checked.
- Click **Add webhook**.



Create Jenkins job

To have Jenkins respond to an event in GitHub such as committing code, create a Jenkins job. Use the URLs for your own GitHub fork.

In your Jenkins website, select **Create new jobs** from the home page:

- Enter *HelloWorld* as job name. Choose **Freestyle project**, then select **OK**.
- Under the **General** section, select **GitHub project** and enter your forked repo URL, such as `https://github.com/cynthn/nodejs-docs-hello-world`
- Under the **Source code management** section, select **Git**, enter your forked repo *.git* URL, such as `https://github.com/cynthn/nodejs-docs-hello-world.git`
- Under the **Build Triggers** section, select **GitHub hook trigger for GITscm polling**.
- Under the **Build** section, choose **Add build step**. Select **Execute shell**, then enter `echo "Test"` in the command window.
- Select **Save** at the bottom of the jobs window.

Test GitHub integration

To test the GitHub integration with Jenkins, commit a change in your fork.

Back in GitHub web UI, select your forked repo, and then select the **index.js** file. Select the pencil icon to edit this file so line 6 reads:

```JavaScript
response.end("Hello World!");
```

To commit your changes, select the **Commit changes** button at the bottom.

In Jenkins, a new build starts under the **Build history** section of the bottom left-hand corner of your job page. Choose the build number link and select **Console output** on the left-hand side. You can view the steps Jenkins takes as your code is pulled from GitHub and the build action outputs the message `Test` to the console. Each time a commit is made in GitHub, the webhook reaches out to Jenkins and triggers a new build in this way.

Define Docker build image

To see the Node.js app running based on your GitHub commits, lets build a Docker image to run the app. The image is built from a Dockerfile that defines how to configure the container that runs the app.

From the SSH connection to your VM, change to the Jenkins workspace directory named after the job you created in a previous step. In this example, that was named *HelloWorld*.

```Bash
cd /var/lib/jenkins/workspace/HelloWorld
```

Create a file in this workspace directory with `sudo sensible-editor Dockerfile` and paste the following contents. Make sure that the whole Dockerfile is copied correctly, especially the first line:

```YAML
FROM node:alpine

EXPOSE 1337

WORKDIR /var/www
```

```
COPY package.json /var/www/
RUN npm install
COPY index.js /var/www/
```

This Dockerfile uses the base Node.js image using Alpine Linux, exposes port 1337 that the Hello World app runs on, then copies the app files and initializes it.

Create Jenkins build rules

In a previous step, you created a basic Jenkins build rule that output a message to the console. Lets create the build step to use our Dockerfile and run the app.

Back in your Jenkins instance, select the job you created in a previous step. Select **Configure** on the left-hand side and scroll down to the **Build** section:

- Remove your existing `echo "Test"` build step. Select the red cross on the top right-hand corner of the existing build step box.
- Choose **Add build step**, then select **Execute shell**
- In the **Command** box, enter the following Docker commands, then select **Save**:

  Bash

  ```
  docker build --tag helloworld:$BUILD_NUMBER .
  docker stop helloworld && docker rm helloworld
  docker run --name helloworld -p 1337:1337 helloworld:$BUILD_NUMBER node
  /var/www/index.js &
  ```

The Docker build steps create an image and tag it with the Jenkins build number so you can maintain a history of images. Any existing containers running the app are stopped and then removed. A new container is then started using the image and runs your Node.js app based on the latest commits in GitHub.

Test your pipeline

To see the whole pipeline in action, edit the *index.js* file in your forked GitHub repo again and select **Commit change**. A new job starts in Jenkins based on the webhook for GitHub. It takes a few seconds to create the Docker image and start your app in a new container.

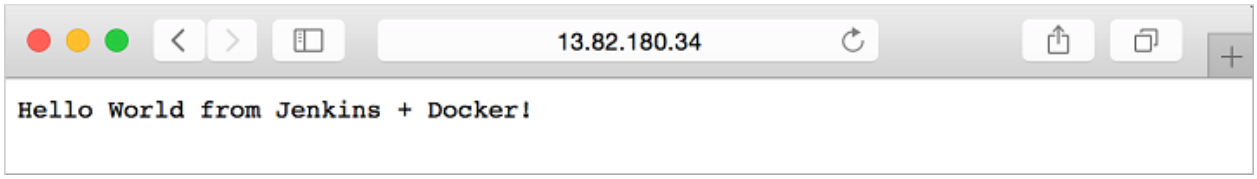
If needed, obtain the public IP address of your VM again:

```
```

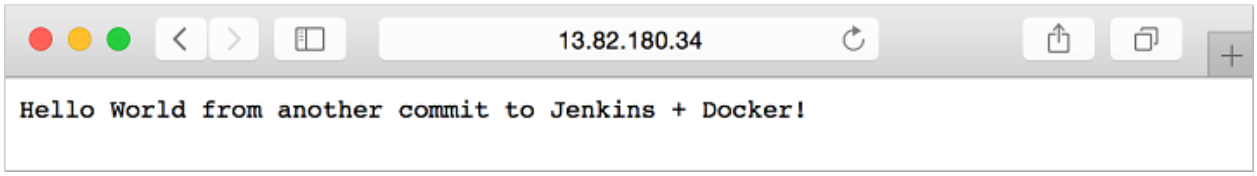
Azure CLI

```
az vm show --resource-group myResourceGroupJenkins --name myVM -d --query
[publicIps] --o tsv
```

Open a web browser and enter `http://<publicIps>:1337`. Your Node.js app is displayed and reflects the latest commits in your GitHub fork as follows:



Now make another edit to the *index.js* file in GitHub and commit the change. Wait a few seconds for the job to complete in Jenkins, then refresh your web browser to see the updated version of your app running in a new container as follows:



Next steps

In this tutorial, you configured GitHub to run a Jenkins build job on each code commit and then deploy a Docker container to test your app. You learned how to:

- ✓ Create a Jenkins VM
- ✓ Install and configure Jenkins
- ✓ Create webhook integration between GitHub and Jenkins
- ✓ Create and trigger Jenkins build jobs from GitHub commits
- ✓ Create a Docker image for your app
- ✓ Verify GitHub commits build new Docker image and updates running app

Advance to the next tutorial to learn more about how to integrate Jenkins with Azure DevOps Services.

Deploy apps with Jenkins and Azure DevOps Services

Tutorial: Deploy to a Service Fabric cluster

Article • 01/22/2021

This tutorial covers several possible ways of setting up your Jenkins environment as well as different ways to deploy your application to a Service Fabric cluster after it has been built. Follow these general steps to successfully configure Jenkins, pull changes from GitHub, build your application, and deploy it to your cluster:

1. Ensure that you install the Prerequisites.
2. Then follow the steps in one of these sections to set up Jenkins:

    - Set up Jenkins inside a Service Fabric cluster,
    - Set up Jenkins outside a Service Fabric cluster, or
    - Install the Service Fabric plug-in in an existing Jenkins environment.

3. After you've set up Jenkins, follow the steps in Create and configure a Jenkins job to set up GitHub to trigger Jenkins when changes are made to your application and to configure your Jenkins job pipeline through the build step to pull the changes from GitHub and build your application.
4. Finally, configure the Jenkins job post-build step to deploy your application to your Service Fabric cluster. There are two ways to configure Jenkins to deploy your application to a cluster:

    - For development and test environments, use Configure deployment using cluster management endpoint. This method is the simplest deployment method to set up.
    - For production environments, use Configure deployment using Azure credentials. Microsoft recommends this method for production environments because with Azure credentials you can limit the access that a Jenkins job has to your Azure resources.

Prerequisites

- Make sure Git is installed locally. You can install the appropriate Git version from the Git downloads page based on your operating system. If you're new to Git, learn more about it from the Git documentation.
- This article uses the *Service Fabric Getting Started Sample* on GitHub: https://github.com/Azure-Samples/service-fabric-java-getting-started for the

application to build and deploy. You can fork this repository to follow along, or, with some modification to the instructions, use your own GitHub project.

Install Service Fabric plug-in in an existing Jenkins environment

If you're adding the Service Fabric plug-in to an existing Jenkins environment, you need to do the following steps:

- Service Fabric CLI (sfctl). Install the CLI at the system level rather than at the user level, so Jenkins can run CLI commands.
- To deploy Java applications, install both Gradle and Open JDK 8.0.
- To deploy .NET Core 2.0 applications, install the .NET Core 2.0 SDK.

After you've installed the prerequisites needed for your environment, you can search for the Azure Service Fabric plug-in in Jenkins marketplace and install it.

After you've installed the plug-in, skip ahead to Create and configure a Jenkins job.

Set up Jenkins inside a Service Fabric cluster

You can set up Jenkins either inside or outside a Service Fabric cluster. The following sections show how to set it up inside a cluster while using an Azure storage account to save the state of the container instance.

1. Ensure that you have a Service Fabric Linux cluster with Docker installed. Service Fabric clusters running in Azure already have Docker installed. If you're running the cluster locally (OneBox dev environment), check if Docker is installed on your machine with the `docker info` command. If it is not installed, install it by using the following commands:

   ```sh
   sudo apt-get install wget
   wget -qO- https://get.docker.io/ | sh
   ```

   > **Note**

Make sure that the 8081 port is specified as a custom endpoint on the cluster. If you are using a local cluster, make sure that port 8081 is open on the host machine and that it has a public-facing IP address.

2. Clone the application, by using the following commands:

```sh
git clone https://github.com/suhuruli/jenkins-container-application.git
cd jenkins-container-application
```

3. Persist the state of the Jenkins container in a file-share:

   a. Create an Azure storage account in the **same region** as your cluster with a name such as `sfjenkinsstorage1`.

   b. Create a **File Share** under the storage Account with a name such as `sfjenkins`.

   c. Click on **Connect** for the file-share and note the values it displays under **Connecting from Linux**, the value should look similar to the one below:

```sh
sudo mount -t cifs
//sfjenkinsstorage1.file.core.windows.net/sfjenkins [mount point] -o
vers=3.0,username=sfjenkinsstorage1,password=
<storage_key>,dir_mode=0777,file_mode=0777
```

> ⓘ **Note**
>
> To mount cifs shares, you need to have the cifs-utils package installed in the cluster nodes.

4. Update the placeholder values in the `setupentrypoint.sh` script with the azure-storage details from step 2.

```sh
vi JenkinsSF/JenkinsOnSF/Code/setupentrypoint.sh
```

   - Replace `[REMOTE_FILE_SHARE_LOCATION]` with the value `//sfjenkinsstorage1.file.core.windows.net/sfjenkins` from the output of the connect in step 2 above.

- Replace `[FILE_SHARE_CONNECT_OPTIONS_STRING]` with the value `vers=3.0,username=sfjenkinsstorage1,password=GB2NPUCQY9LDGeG9Bci5dJV91T6SrA7OxrYBUsFHyueR62viMrC6NIzyQLCKNz0o7pepGfGY+vTa9gxzEtfZHw==,dir_mode=0777,file_mode=0777` from step 2 above.

5. **Secure Cluster Only:**

   To configure the deployment of applications on a secure cluster from Jenkins, the cluster certificate must be accessible within the Jenkins container. In the *ApplicationManifest.xml* file, under the **ContainerHostPolicies** tag add this certificate reference and update the thumbprint value with that of the cluster certificate.

   ```
   XML

   <CertificateRef Name="MyCert" X509FindValue="[Thumbprint]"/>
   ```

   Additionally, add the following lines under the **ApplicationManifest** (root) tag in the *ApplicationManifest.xml* file and update the thumbprint value with that of the cluster certificate.

   ```
   XML

   <Certificates>
     <SecretsCertificate X509FindType="FindByThumbprint" X509FindValue="
   [Thumbprint]" />
   </Certificates>
   ```

6. Connect to the cluster and install the container application.

   Secure Cluster

   ```
   sh

   sfctl cluster select --endpoint https://PublicIPorFQDN:19080  --pem
   [Pem] --no-verify # cluster connect command
   bash Scripts/install.sh
   ```

   The preceding command takes the certificate in PEM format. If your certificate is in PFX format, you can use the following command to convert it. If your PFX file isn't password protected, specify the **passin** parameter as `-passin pass:`.

   ```
   sh
   ```

```
openssl pkcs12 -in cert.pfx -out cert.pem -nodes -passin
pass:MyPassword1234!
```

Unsecure Cluster

sh

```
sfctl cluster select --endpoint http://PublicIPorFQDN:19080 # cluster
connect command
bash Scripts/install.sh
```

This installs a Jenkins container on the cluster, and can be monitored by using the Service Fabric Explorer.

> ⓘ **Note**
>
> It may take a couple of minutes for the Jenkins image to be downloaded on the cluster.

7. From your browser, go to `http://PublicIPorFQDN:8081`. It provides the path of the initial admin password required to sign in.

8. Look at the Service Fabric Explorer to determine on which node the Jenkins container is running. Secure Shell (SSH) sign in to this node.

sh

```
ssh user@PublicIPorFQDN -p [port]
```

9. Get the container instance ID by using `docker ps -a`.

10. Secure Shell (SSH) sign in to the container and paste the path you were shown on the Jenkins portal. For example, if in the portal it shows the path `PATH_TO_INITIAL_ADMIN_PASSWORD`, run the following commands:

sh

```
docker exec -t -i [first-four-digits-of-container-ID] /bin/bash   #
This takes you inside Docker shell
```

sh

```
cat PATH_TO_INITIAL_ADMIN_PASSWORD # This displays the password value
```

11. On the Jenkins Getting Started page, choose the Select plug-in to install option, select the **None** checkbox, and click install.

12. Create a user or select to continue as an admin.

After you've set up Jenkins, skip ahead to Create and configure a Jenkins job.

Set up Jenkins outside a Service Fabric cluster

You can set up Jenkins either inside or outside of a Service Fabric cluster. The following sections show how to set it up outside a cluster.

1. Make sure that Docker is installed on your machine by running `docker info` in the terminal. The output indicates if the Docker service is running.

2. If Docker is not installed, run the following commands:

```sh
sudo apt-get install wget
wget -qO- https://get.docker.io/ | sh
```

3. Pull the Service Fabric Jenkins container image: `docker pull rapatchi/jenkins:latest`. This image comes with Service Fabric Jenkins plug-in pre-installed.

4. Run the container image: `docker run -itd -p 8080:8080 rapatchi/jenkins:latest`

5. Get the ID of the container image instance. You can list all the Docker containers with the command `docker ps –a`

6. Sign in to the Jenkins portal with the following steps:

   a. Sign in to a Jenkins shell from your host. Use the first four digits of the container ID. For example, if the container ID is `2d24a73b5964`, use `2d24`.

   ```sh
   docker exec -it [first-four-digits-of-container-ID] /bin/bash
   ```

   b. From the Jenkins shell, get the admin password for your container instance:

```sh
cat /var/jenkins_home/secrets/initialAdminPassword
```

c. To sign in to the Jenkins dashboard, open the following URL in a web browser: `http://<HOST-IP>:8080`. Use the password from the previous step to unlock Jenkins.

d. (Optional.) After you sign in for the first time, you can create your own user account and use that for the following steps, or you can continue to use the administrator account. If you create a user, you need to continue with that user.

7. Set up GitHub to work with Jenkins by using the steps in Generating a new SSH key and adding it to the SSH agent.

   - Use the instructions provided by GitHub to generate the SSH key, and to add the SSH key to the GitHub account that is hosting the repository.
   - Run the commands mentioned in the preceding link in the Jenkins Docker shell (and not on your host).
   - To sign in to the Jenkins shell from your host, use the following command:

     ```sh
     docker exec -t -i [first-four-digits-of-container-ID] /bin/bash
     ```

Make sure that the cluster or machine where the Jenkins container image is hosted has a public-facing IP address. This enables the Jenkins instance to receive notifications from GitHub.

After you've set up Jenkins, continue on to the next section, Create and configure a Jenkins job.

Create and configure a Jenkins job

The steps in this section show you how to configure a Jenkins job to respond to changes in a GitHub repo, fetch the changes, and build them. At the end of this section, you're directed to the final steps to configure the job to deploy your application based on whether you're deploying to a development/test environment or to a production environment.

1. On the Jenkins dashboard, click **New Item**.

2. Enter an item name (for example, **MyJob**). Select **free-style project**, and click **OK**.

3. The Job configuration page opens. (To get to the configuration from the Jenkins dashboard, click the job, and then click **Configure**).

4. On the **General** tab, check the box for **GitHub project**, and specify your GitHub project URL. This URL hosts the Service Fabric Java application that you want to integrate with the Jenkins continuous integration, continuous deployment (CI/CD) flow (for example, `https://github.com/{your-github-account}/service-fabric-java-getting-started`).

5. On the **Source Code Management** tab, select **Git**. Specify the repository URL that hosts the Service Fabric Java application that you want to integrate with the Jenkins CI/CD flow (for example, `https://github.com/{your-github-account}/service-fabric-java-getting-started`). You can also specify which branch to build (for example, `/master`).

6. Configure your *GitHub* repository to talk to Jenkins:

   a. On your GitHub repository page, go to **Settings** > **Integrations and Services**.

   b. Select **Add Service**, type **Jenkins**, and select the **Jenkins-GitHub plug-in**.

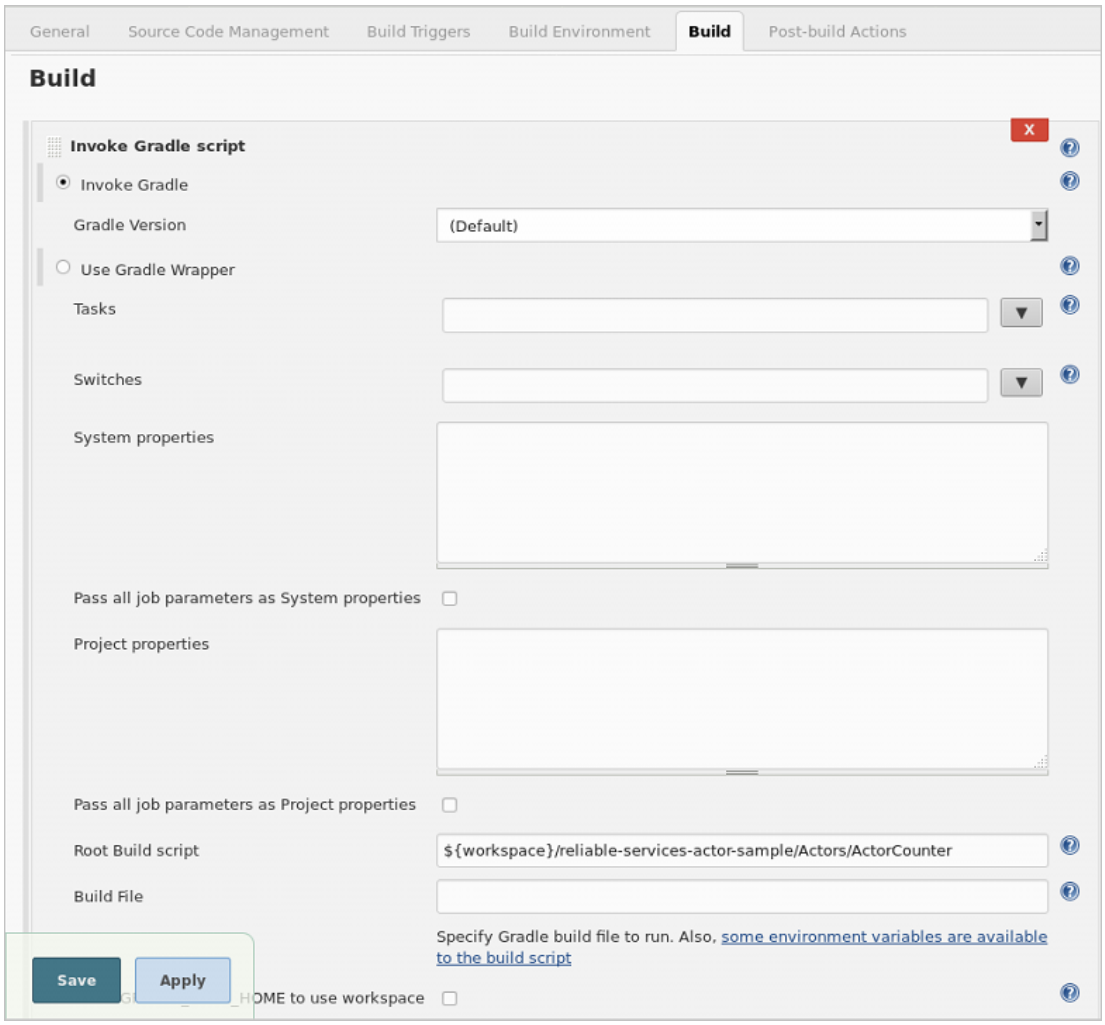
   c. Enter your Jenkins webhook URL (by default, it should be `http://<PublicIPorFQDN>:8081/github-webhook/`). Click **add/update service**.

   d. A test event is sent to your Jenkins instance. You should see a green check by the webhook in GitHub, and your project will build.

7. On the **Build Triggers** tab in Jenkins, select which build option you want. For this example, you want to trigger a build whenever a push to the repository happens, so select **GitHub hook trigger for GITScm polling**. (Previously, this option was called **Build when a change is pushed to GitHub**.)

8. On the **Build** tab, do one of the following depending on whether you're building a Java application or a .NET Core application:

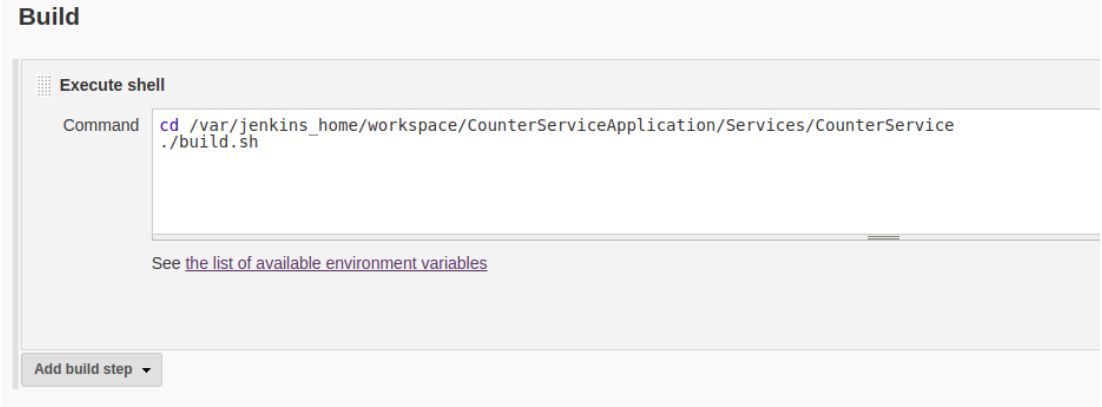
   - **For Java Applications:** From the **Add build step** drop-down, select **Invoke Gradle Script**. Click **Advanced**. In the advanced menu, specify the path to **Root build script** for your application. It picks up build.gradle from the path specified and works accordingly. For the ActorCounter application, this is: `${WORKSPACE}/reliable-services-actor-sample/Actors/ActorCounter`.



- **For .NET Core Applications:** From the **Add build step** drop-down, select **Execute Shell**. In the command box that appears, the directory first needs to be changed to the path where the `build.sh` file is located. Once the directory has been changed, the `build.sh` script can be run to build the application.

```sh
cd /var/jenkins_home/workspace/[Job Name]/[Path to build.sh]
./build.sh
```

The following screenshot shows an example of the commands that are used to build the Counter Service sample with a Jenkins job name of `CounterServiceApplication`.



9. To configure Jenkins to deploy your app to a Service Fabric cluster in the post-build actions, you need the location of that cluster's certificate in your Jenkins container. Choose one of the following depending on whether your Jenkins container is running inside or outside of your cluster and note the location of the cluster certificate:

    - **For Jenkins running inside your cluster:** The path to the certificate can be found by echoing the value of the *Certificates_JenkinsOnSF_Code_MyCert_PEM* environment variable from within the container.

      ```sh
      echo $Certificates_JenkinsOnSF_Code_MyCert_PEM
      ```

    - **For Jenkins running outside your cluster:** Follow these steps to copy the cluster certificate to your container:

        a. Your certificate must be in PEM format. If you don't have a PEM file, you can create one from the certificate PFX file. If your PFX file is not password protected, run the following command from your host:

        ```sh
        openssl pkcs12 -in clustercert.pfx -out clustercert.pem -nodes
        -passin pass:
        ```

        If the PFX file is password protected, include the password in the `-passin` parameter. For example:

        ```sh
        openssl pkcs12 -in clustercert.pfx -out clustercert.pem -nodes
        -passin pass:MyPassword1234!
        ```

b. To get the container ID for your Jenkins container, run `docker ps` from your host.

c. Copy the PEM file to your container with the following Docker command:

```sh
docker cp clustercert.pem [first-four-digits-of-container-ID]:/var/jenkins_home
```

You're almost finished! Keep the Jenkins job open. The only remaining task is to configure the post-build steps to deploy your application to your Service Fabric cluster:

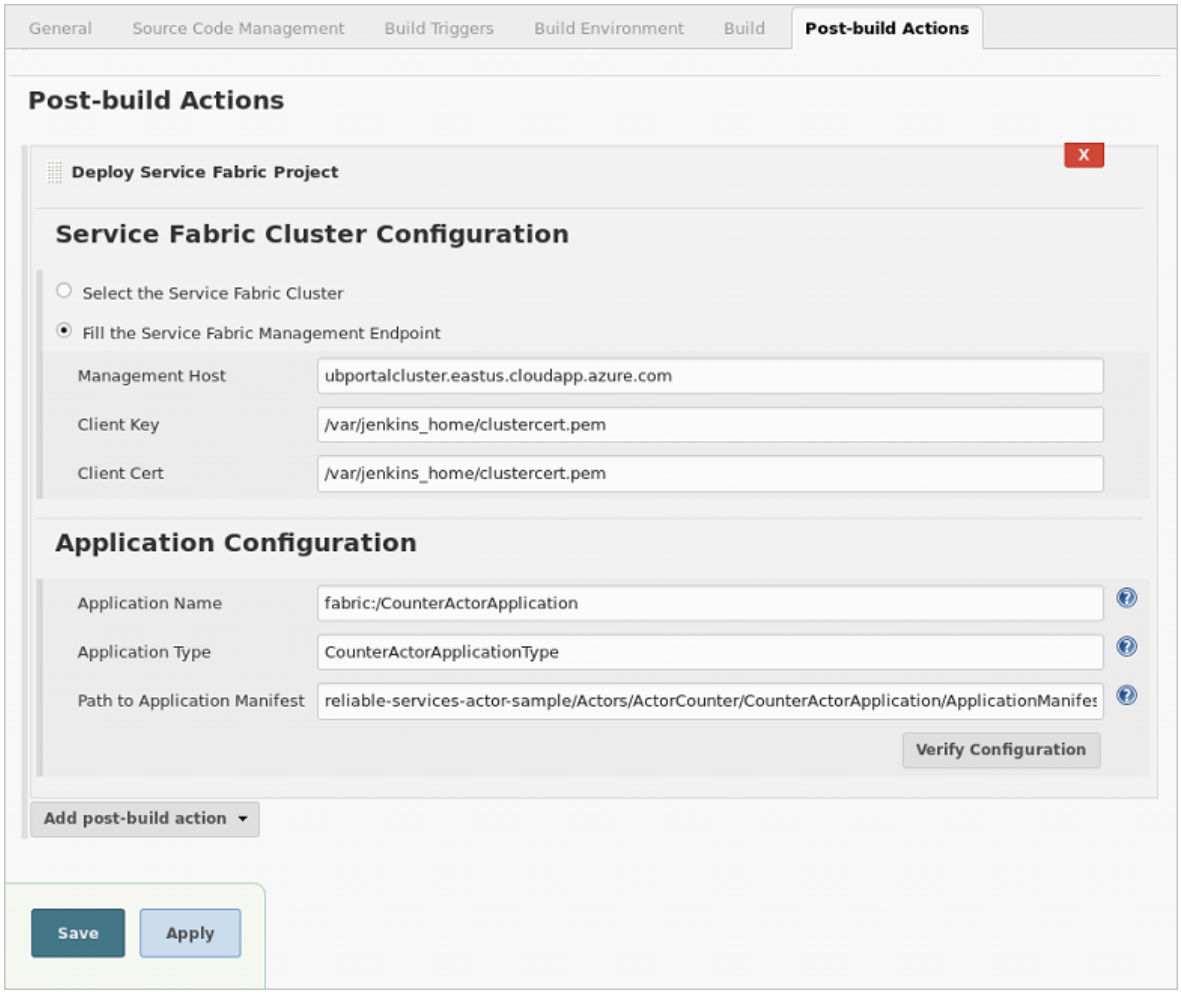
- To deploy to a development or test environment, follow the steps in Configure deployment using cluster management endpoint.
- To deploy to a production environment, follow the steps in Configure deployment using Azure credentials.

Configure deployment using cluster management endpoint

For development and test environments, you can use the cluster management endpoint to deploy your application. Configuring the post-build action with the cluster management endpoint to deploy your application requires the least amount of set-up. If you're deploying to a production environment, skip ahead to Configure deployment using Azure credentials to configure a Microsoft Entra service principal to use during deployment.

1. In the Jenkins job, click the **Post-build Actions** tab.
2. From the **Post-Build Actions** drop-down, select **Deploy Service Fabric Project**.
3. Under **Service Fabric Cluster Configuration**, select the **Fill the Service Fabric Management Endpoint** radio button.
4. For **Management Host**, enter the connection endpoint for your cluster; for example `{your-cluster}.eastus.cloudapp.azure.com`.
5. For **Client Key** and **Client Cert**, enter the location of the PEM file in your Jenkins container; for example `/var/jenkins_home/clustercert.pem`. (You copied the location of the certificate the last step of Create and configure a Jenkins job.)

6. Under **Application Configuration**, configure the **Application Name**, **Application Type**, and the (relative) **Path to Application Manifest** fields.



7. Click **Verify Configuration**. On successful verification, click **Save**. Your Jenkins job pipeline is now fully configured. Skip ahead to Next steps to test your deployment.

Configure deployment using Azure credentials

For production environments, configuring an Azure credential to deploy your application is strongly recommended. This section shows you how to configure a Microsoft Entra service principal to use to deploy your application in the post-build action. You can assign service principals to roles in your directory to limit the permissions of the Jenkins job.

For development and test environments, you can configure either Azure credentials or the cluster management endpoint to deploy your application. For details about how to configure a cluster management endpoint, see Configure deployment using cluster management endpoint.

1. To create a Microsoft Entra service principal and assign it permissions in your Azure subscription, follow the steps in Use the portal to create a Microsoft Entra

application and service principal. Pay attention to the following:

- While following the steps in the topic, be sure to copy and save the following values: *Application ID*, *Application key*, *Directory ID (Tenant ID)*, and *Subscription ID*. You need them to configure the Azure credentials in Jenkins.
- If you don't have the required permissions on your directory, you'll need to ask an administrator to either grant you the permissions or create the service principal for you, or you'll need to configure the management endpoint for your cluster in the **Post-Build Actions** for your job in Jenkins.
- In the Create a Microsoft Entra application section, you can enter any well-formed URL for the **Sign-on URL**.
- In the Assign application to a Role section, you can assign your application the *Reader* role on the resource group for your cluster.

2. Back in the Jenkins job, click the **Post-build Actions** tab.

3. From the **Post-Build Actions** drop-down, select **Deploy Service Fabric Project**.

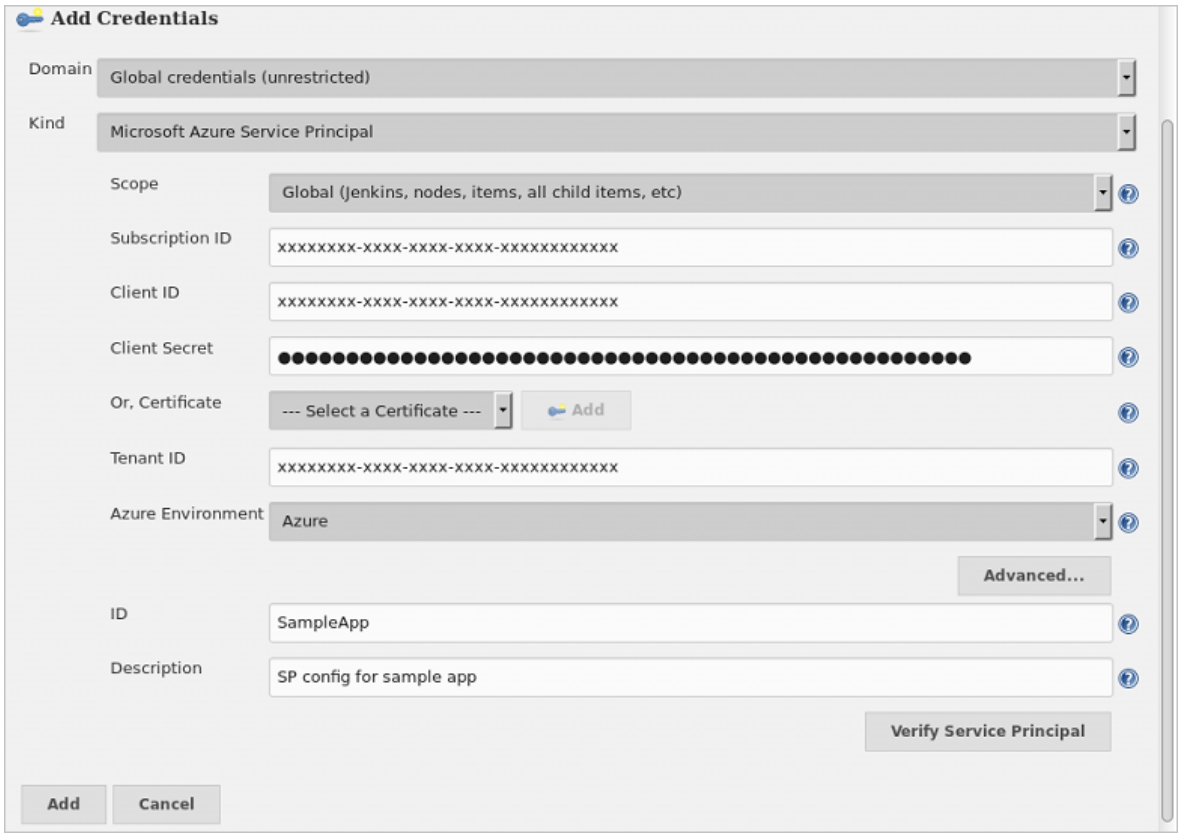
4. Under **Service Fabric Cluster Configuration**, Click **Select the Service Fabric Cluster**. Click **Add** next to **Azure Credentials**. Click **Jenkins** to select the Jenkins Credentials Provider.

5. In the Jenkins Credentials Provider, select **Microsoft Azure Service Principal** from the **Kind** drop-down.

6. Use the values you saved when setting up your service principal in Step 1 to set the following fields:

   - **Client ID**: *Application ID*
   - **Client Secret**: *Application key*
   - **Tenant ID**: *Directory ID*
   - **Subscription ID**: *Subscription ID*

7. Enter a descriptive **ID** that you use to select the credential in Jenkins and a brief **Description**. Then click **Verify Service Principal**. If the verification succeeds, click **Add**.



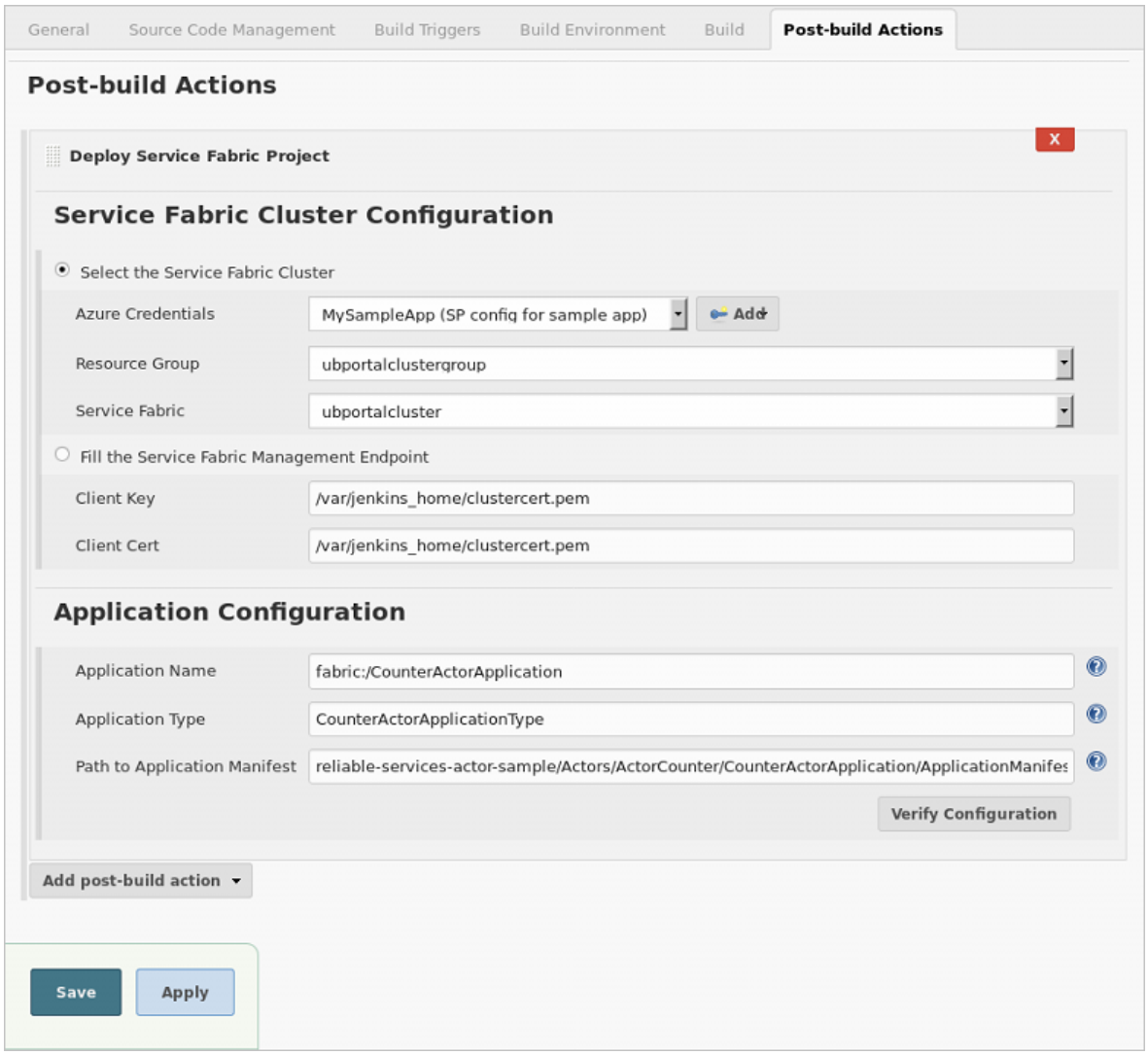
8. Back under **Service Fabric Cluster Configuration**, make sure that your new credential is selected for **Azure Credentials**.

9. From the **Resource Group** drop-down, select the resource group of the cluster you want to deploy the application to.

10. From the **Service Fabric** drop-down, select the cluster that you want to deploy the application to.

11. For **Client Key** and **Client Cert**, enter the location of the PEM file in your Jenkins container. For example `/var/jenkins_home/clustercert.pem`.

12. Under **Application Configuration**, configure the **Application Name**, **Application Type**, and the (relative) **Path to Application Manifest** fields.



13. Click **Verify Configuration**. On successful verification, click **Save**. Your Jenkins job pipeline is now fully configured. Continue on to Next steps to test your deployment.

Troubleshooting the Jenkins plug-in

If you encounter any bugs with the Jenkins plug-in, file an issue in the Jenkins JIRA for the specific component.

Ideas to try

GitHub and Jenkins are now configured. Consider making some sample change in the `reliable-services-actor-sample/Actors/ActorCounter` project in your fork of the repository, https://github.com/Azure-Samples/service-fabric-java-getting-started. Push your changes to the remote `master` branch (or any branch that you have configured to work with). This triggers the Jenkins job, `MyJob`, that you configured. It fetches the changes from GitHub, builds them, and deploys the application to the cluster you specified in post-build actions.

Next steps

Jenkins on Azure

Jenkins plug-ins for Azure

Article • 11/07/2023

> **Important**
>
> Many Azure services have Jenkins plug-ins. Some of these plug-ins will be out of support as of February 29, 2024. Azure CLI is the currently recommended way to integrate Jenkins with Azure services. For more information, refer to the article **Jenkins plug-ins for Azure**.

Many Azure services and features are accessible via Jenkins plug-ins. These services support an array of possibilities regarding continuous integration and continuous deployment (CI/CD) for your DevOps environment. Click on any of the Jenkins plug-ins to learn more about that plug-in.

Jenkins plug-in	Description	Current status	Community Supported
Azure Service Fabric	Jenkins plug-in for Linux Azure Service Fabric projects.		
Microsoft Entra ID	Jenkins plug-in that supports authentication & authorization via Microsoft Entra ID.	Retiring on February 29, 2024	Yes
Azure App Service	Jenkins plug-in to deploy an Azure App Service (currently supports only Web App)	Retiring on February 29, 2024	
Azure Artifact Manager	Azure Artifact Manager plug-in is an Artifact Manager that allows you store your artifacts into Azure Blob Storage. Azure Artifact Manager plug-in works transparently to Jenkins and your jobs, it is like the default Artifact Manager.	Retiring on February 29, 2024	Yes
Azure Container Agents	Azure Container Agents plug-in can help you to run a container as an agent in Jenkins	Retiring on February 29, 2024	Yes
Azure Container Registry Task	Jenkins plug-in to send a docker-build request to Azure Container Registry.	Retiring on February 29, 2024	

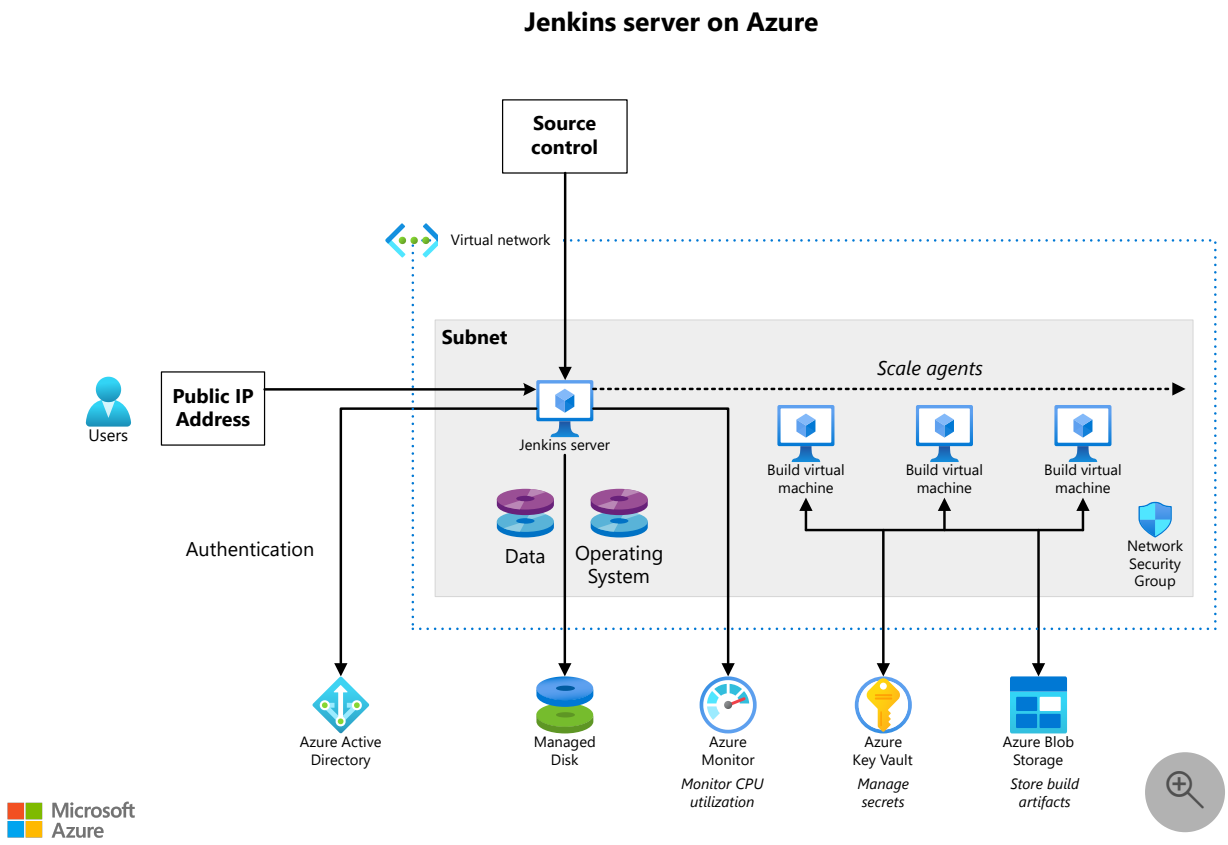
Jenkins plug-in	Description	Current status	Community Supported
Azure Container Service	Jenkins plug-in to deploy configurations to Azure Container Service (AKS).	Retiring on February 29, 2024	
Azure Credential	Jenkins plug-in to manage Azure credentials.	Retiring on February 29, 2024	Yes
Azure Function	Jenkins plug-in to deploy an Azure Function.	Retiring on February 29, 2024	
Azure IoT Edge	Azure IoT Edge plug-in makes it easy to set up a CI/CD pipeline for developing an IoT Edge solution on Jenkins.	Retiring on February 29, 2024	
Azure Storage	plug-in for uploading build artifacts to, or downloading build dependencies from, Microsoft Azure Blob storage.	Retiring on February 29, 2024	Yes
Azure virtual machine scale set	Jenkins plug-in to deploy VM images to Azure virtual machine scale sets.	Retiring on February 29, 2024	
Azure VM agents	Jenkins plug-in to create Jenkins agents in Azure virtual machines (via Azure Resource Manager template).	Retiring on February 29, 2024	Yes

Run a Jenkins server on Azure

Microsoft Entra ID | Azure Blob Storage | Azure Key Vault | Azure Monitor | Azure Virtual Machines

This scenario explains the architecture and considerations to take into account when installing and configuring Jenkins.

Architecture



Download a Visio file of this architecture.

Workflow

The architecture consists of the following aspects:

- **Resource group.** A resource group is used to group Azure assets so they can be managed by lifetime, owner, and other criteria. Use resource groups to deploy and monitor Azure assets as a group and track billing costs by resource group. You can also delete resources as a set, which is useful for test deployments.

- **Jenkins server.** A virtual machine is deployed to run Jenkins as an automation server and serves as Jenkins Primary. In the section Install and configure Jenkins, you'll see how to install Jenkins on a new VM.

  > ⓘ Note
  >
  > Nginx is installed on the VM to act as a reverse proxy to Jenkins. You can configure Nginx to enable SSL for the Jenkins server.

- **Virtual network.** A virtual network connects Azure resources to each other and provides logical isolation. In this architecture, the Jenkins server runs in a virtual network.
- **Subnets.** The Jenkins server is isolated in a subnet to make it easier to manage and segregate network traffic without affecting performance.
- **Network security groups.** Use network security groups to restrict network traffic from the Internet to the subnet of a virtual network.
- **Managed disks.** A managed disk is a persistent virtual hard disk (VHD) used for application storage and also to maintain the state of the Jenkins server and provide disaster recovery. Data disks are stored in Azure Storage. For high performance, premium storage is recommended.
- **Azure Blob storage.** The Microsoft Azure Storage Learn how to Azure Blob storage to store the build artifacts that are created and shared with other Jenkins builds.
- **Microsoft Entra ID.** Microsoft Entra ID supports user authentication, allowing you to set up SSO. Microsoft Entra service principals define the policy and permissions for each role authorization in the workflow, using Azure role-based access control (Azure RBAC). Each service principal is associated with a Jenkins job.
- **Azure Key Vault.** To manage secrets and cryptographic keys used to provision Azure resources when secrets are required, this architecture uses Azure Key Vault.
- **Azure monitoring services**. This service monitors the Azure virtual machine hosting Jenkins. This deployment monitors the virtual machine status and CPU utilization and sends alerts.

Scenario details

This architecture supports disaster recovery with Azure services but does not cover more advanced scale-out scenarios involving multiple primaries or high availability (HA) with

no downtime. For general insights about the various Azure components, including a step-by-step tutorial about building out a CI/CD pipeline on Azure, see Jenkins on Azure.

The focus of this document is on the core Azure operations needed to support Jenkins, including the use of Azure Storage to maintain build artifacts, the security items needed for SSO, other services that can be integrated, and scalability for the pipeline. The architecture is designed to work with an existing source control repository. For example, a common scenario is to start Jenkins jobs based on GitHub commits.

Potential use cases

- Automate CI/CD pipelines
- Ensure high availability (HA) for mission-critical services

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see Microsoft Azure Well-Architected Framework.

Scalability

Jenkins can dynamically scale to support workloads as needed. For elastic builds, do not run builds on the Jenkins primary server. Instead, offload build tasks to Jenkins agents, which can be elastically scaled in and out as need. Consider two options for scaling agents:

- Scale Jenkins deployments with Azure VM Agents. VM Agents enable elastic scale-out for agents and can use distinct types of virtual machines. You can specify a different base image from Azure Marketplace or use a custom image. For details about how the Jenkins agents scale, see Architecting for Scale in the Jenkins documentation.
- Run a container as an agent in either Azure Container Service with Kubernetes, or Azure Container Instances.

Virtual machines generally cost more to scale than containers. To use containers for scaling, however, your build process must run with containers.

Also, use Azure Storage to share build artifacts that may be used in the next stage of the pipeline by other build agents.

Scale the Jenkins server

When you create a VM and install Jenkins, you can specify the size of the VM. Selecting the correct VM server size depends on the size of the expected workload. The Jenkins community maintains a selection guide to help identify the configuration that best meets your requirements. Azure offers many sizes for Linux VMs to meet any requirements. For more information about scaling the Jenkins primary, see the Jenkins community of best practices, which also includes details about scaling Jenkins.

Availability

Availability in the context of a Jenkins server means being able to recover any state information associated with your workflow, such as test results, libraries you have created, or other artifacts. Critical workflow state or artifacts must be maintained to recover the workflow if the Jenkins server goes down. To assess your availability requirements, consider two common metrics:

- Recovery Time Objective (RTO) specifies how long you can go without Jenkins.
- Recovery Point Objective (RPO) indicates how much data you can afford to lose if a disruption in service affects Jenkins.

In practice, RTO, and RPO imply redundancy and backup. Availability is not a question of hardware recovery - that is part of Azure - but rather ensuring you maintain the state of your Jenkins server. Microsoft offers a service level agreement (SLA) for single VM instances. If this SLA doesn't meet your uptime requirements, make sure you have a plan for disaster recovery, or consider using a multi-primary Jenkins server deployment (not covered in this document).

Consider using the disaster recovery scripts in step 7 of the deployment to create an Azure Storage account with managed disks to store the Jenkins server state. If Jenkins goes down, it can be restored to the state stored in this separate storage account.

Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see Overview of the security pillar.

Use the following approaches to help lock down security on a basic Jenkins server, since in its basic state, it is not secure.

- Set up a secure way to log into the Jenkins server. This architecture uses HTTP and has a public IP, but HTTP is not secure by default. Consider setting up HTTPS on

the Nginx server being used for a secure logon.

> **Note**
>
> When adding SSL to your server, create a network security group rule for the Jenkins subnet to open port 443. For more information, see **How to open ports to a virtual machine with the Azure portal**.

- Ensure that the Jenkins configuration prevents cross site request forgery (Manage Jenkins > Configure Global Security). This option is the default for Microsoft Jenkins Server.
- Configure read-only access to the Jenkins dashboard by using the Matrix Authorization Strategy Plugin.
- Use Azure RBAC to restrict the access of the service principal to the minimum required to run the jobs. This level of security helps limit the scope of damage from a rogue job.

Jenkins jobs often require secrets to access Azure services that require authorization, such as Azure Container Service. Use Key Vault to manage these secrets securely. Use Key Vault to store service principal credentials, passwords, tokens, and other secrets.

To get a central view of the security state of your Azure resources, use Microsoft Defender for Cloud. Defender for Cloud monitors potential security issues and provides a comprehensive picture of the security health of your deployment. Defender for Cloud is configured per Azure subscription. Enable security data collection as described in the Microsoft Defender for Cloud quick start guide. When data collection is enabled, Defender for Cloud automatically scans any virtual machines created under that subscription.

The Jenkins server has its own user management system, and the Jenkins community provides best practices for securing a Jenkins instance on Azure.

Manageability

Use resource groups to organize the Azure resources that are deployed. Deploy production environments and development/test environments in separate resource groups, so that you can monitor each environment's resources and roll up billing costs by resource group. You can also delete resources as a set, which is useful for test deployments.

Azure provides several features for monitoring and diagnostics of the overall infrastructure. To monitor CPU usage, this architecture deploys Azure Monitor. For example, you can use Azure Monitor to monitor CPU utilization, and send a notification if CPU usage exceeds 80 percent. (High CPU usage indicates that you might want to scale up the Jenkins server VM.) You can also notify a designated user if the VM fails or becomes unavailable.

Deploy this solution

Install and configure Jenkins

To create a VM and install Jenkins, follow the instructions in the article, Quickstart: Configure Jenkins using Azure CLI.

Next steps

The following online communities can answer questions and help you configure a successful deployment:

- Jenkins Community Blog
- Azure Forum
- Stack Overflow Jenkins

Related resources

- Design a CI/CD pipeline using Azure DevOps
- Container CI/CD using Jenkins and Kubernetes on Azure Kubernetes Service
- Immutable Infrastructure CI/CD using Jenkins and Terraform on Azure Virtual Architecture overview
- Java CI/CD using Jenkins and Azure Web Apps