User Handbook

Tutorials

Resources

Build a Java app with Maven

Table of Contents

This tutorial shows you how to use Jenkins to orchestrate building a simple Java application with Maven.

If you are a Java developer who uses Maven and who is new to CI/CD concepts, or you might be familiar with these concepts but don't know how to implement building your application using Jenkins, then this tutorial is for you.

The simple Java application (which you'll obtain from a sample repository on GitHub) outputs the string "Hello world!" and is accompanied by a couple of unit tests to check that the main application works as expected. The results of these tests are saved to a JUnit XML report.

Duration: This tutorial takes 20-40 minutes to complete (assuming you've already met the prerequisites below). The exact duration will depend on the speed of your machine and whether or not you've already run Jenkins in Docker from another tutorial.

You can stop this tutorial at any point in time and continue from where you left off.

If you've already run through another tutorial, you can skip the Prerequisites and Run Jenkins in Docker sections below and proceed on to forking the sample repository. (Just ensure you have Git installed locally.) If you need to restart Jenkins, simply follow the restart instructions in Stopping and restarting Jenkins and then proceed on.

Prerequisites

For this tutorial, you will require:

- A macOS, Linux or Windows machine with:
  - 256 MB of RAM, although more than 512MB is recommended.
  - 10 GB of drive space for Jenkins and your Docker images and containers.
- The following software installed:
  - Docker - Read more about installing Docker in the Installing Docker section of the Installing Jenkins page.
    Note: If you use Linux, this tutorial assumes that you are not running Docker commands as the

root user, but instead with a single user account that also has access to the other tools used throughout this tutorial.

- Git and optionally GitHub Desktop.

Run Jenkins in Docker

In this tutorial, you'll be running Jenkins as a Docker container from the `jenkins/jenkins` Docker image.

To run Jenkins in Docker, follow the relevant instructions below for either macOS and Linux or Windows.

You can read more about Docker container and image concepts in the Docker section of the Installing Jenkins page.

On macOS and Linux

1. Open up a terminal window.
2. Create a bridge network in Docker using the following `docker network create` command:

   ```
   docker network create jenkins
   ```

3. In order to execute Docker commands inside Jenkins nodes, download and run the `docker:dind` Docker image using the following link:https://docs.docker.com/engine/reference/run/ [`docker run`] command:

   ```
   docker run \
     --name jenkins-docker \(1)
     --rm \(2)
     --detach \(3)
     --privileged \(4)
     --network jenkins \(5)
     --network-alias docker \(6)
     --env DOCKER_TLS_CERTDIR=/certs \(7)
     --volume jenkins-docker-certs:/certs/client \(8)
     --volume jenkins-data:/var/jenkins_home \(9)
     --publish 3000:3000 \(10)
     --publish 2376:2376 \(11)
     docker:dind(12)
   ```

   1 (*Optional*) Specifies the Docker container name to use for running the image. By default, Docker will generate a unique name for the container.

   2 (*Optional*) Automatically removes the Docker container (the instance of the Docker image) when it is shut down.

   3 (*Optional*) Runs the Docker container in the background. This instance can be stopped later by running `docker stop jenkins-docker`.

   4 Running Docker in Docker currently requires privileged access to function properly. This requirement may be relaxed with newer Linux kernel versions.

   5 This corresponds with the network created in the earlier step.

   6 Makes the Docker in Docker container available as the hostname `docker` within the `jenkins` network.

   7 Enables the use of TLS in the Docker server. Due to the use of a privileged container, this is recommended, though it requires the use of the shared volume described below. This environment variable controls the root directory where Docker TLS certificates are managed.

   8 Maps the `/certs/client` directory inside the container to a Docker volume named `jenkins-docker-certs` as created above.

   9 Maps the `/var/jenkins_home` directory inside the container to the Docker volume named `jenkins-data`. This will allow for other Docker containers controlled by this Docker container's

Docker daemon to mount data from Jenkins.

10 Exposes the Docker daemon port, used by some of tutorials.

11 (*Optional*) Exposes the Docker daemon port on the host machine. This is useful for executing `docker` commands on the host machine to control this inner Docker daemon.

12 The `docker:dind` image itself. This image can be downloaded before running by using the command: `docker image pull docker:dind`.

Note: If copying and pasting the command snippet above does not work, try copying and pasting this annotation-free version here:

```
docker run --name jenkins-docker --rm --detach \
  --privileged --network jenkins --network-alias docker \
  --env DOCKER_TLS_CERTDIR=/certs \
  --volume jenkins-docker-certs:/certs/client \
  --volume jenkins-data:/var/jenkins_home \
  --publish 3000:3000 \
  --publish 2376:2376 docker:dind
```

4. Customise official Jenkins Docker image, by executing below two steps:

   a. Create Dockerfile with the following content:

   ```
   FROM jenkins/jenkins:2.263.4-lts-jdk11
   USER root
   RUN apt-get update && apt-get install -y apt-transport-https \
          ca-certificates curl gnupg2 \
          software-properties-common
   RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
   RUN apt-key fingerprint 0EBFCD88
   RUN add-apt-repository \
          "deb [arch=amd64] https://download.docker.com/linux/debian \
          $(lsb_release -cs) stable"
   RUN apt-get update && apt-get install -y docker-ce-cli
   USER jenkins
   RUN jenkins-plugin-cli --plugins blueocean:1.24.4
   ```

   b. Build a new docker image from this Dockerfile and assign the image a meaningful name, e.g. "myjenkins-blueocean:1.1":

   ```
   docker build -t myjenkins-blueocean:1.1 .
   ```

   Keep in mind that the process described above will automatically download the official Jenkins Docker image if this hasn't been done before.

5. Run your own `myjenkins-blueocean:1.1` image as a container in Docker using the following `docker run` command:

```
docker run \
  --name jenkins-blueocean \(1)
  --rm \(2)
  --detach \(3)
  --network jenkins \(4)
  --env DOCKER_HOST=tcp://docker:2376 \(5)
  --env DOCKER_CERT_PATH=/certs/client \
  --env DOCKER_TLS_VERIFY=1 \
  --publish 8080:8080 \(6)
  --publish 50000:50000 \(7)
  --volume jenkins-data:/var/jenkins_home \(8)
  --volume jenkins-docker-certs:/certs/client:ro \(9)
  --volume "$HOME":/home \(10)
  myjenkins-blueocean:1.1 (11)
```

1 (*Optional*) Specifies the Docker container name for this instance of the Docker image.

2 (*Optional*) Automatically removes the Docker container when it is shut down.

3 (*Optional*) Runs the current container in the background (i.e. "detached" mode) and outputs the container ID. If you do not specify this option, then the running Docker log for this container is output in the terminal window.

4 Connects this container to the `jenkins` network defined in the earlier step. This makes the Docker daemon from the previous step available to this Jenkins container through the hostname `docker`.

5 Specifies the environment variables used by `docker`, `docker-compose`, and other Docker tools to connect to the Docker daemon from the previous step.

6 Maps (i.e. "publishes") port 8080 of the current container to port 8080 on the host machine. The first number represents the port on the host while the last represents the container's port. Therefore, if you specified `-p 49000:8080` for this option, you would be accessing Jenkins on your host machine through port 49000.

7 (*Optional*) Maps port 50000 of the current container to port 50000 on the host machine. This is only necessary if you have set up one or more inbound Jenkins agents on other machines, which in turn interact with your `jenkins-blueocean` container (the Jenkins "controller"). Inbound Jenkins agents communicate with the Jenkins controller through TCP port 50000 by default. You can change this port number on your Jenkins controller through the Configure Global Security page. If you were to change the **TCP port for inbound Jenkins agents** of your Jenkins controller to 51000 (for example), then you would need to re-run Jenkins (via this `docker run …` command) and specify this "publish" option with something like `--publish 52000:51000`, where the last value matches this changed value on the Jenkins controller and the first value is the port number on the machine hosting the Jenkins controller. Inbound Jenkins agents communicate with the Jenkins controller on that port (52000 in this example). Note that WebSocket agents do not need this configuration.

8 Maps the `/var/jenkins_home` directory in the container to the Docker volume with the name `jenkins-data`. Instead of mapping the `/var/jenkins_home` directory to a Docker volume, you could also map this directory to one on your machine's local file system. For example, specifying the option `--volume $HOME/jenkins:/var/jenkins_home` would map the container's `/var/jenkins_home` directory to the `jenkins` subdirectory within the `$HOME` directory on your local machine, which would typically be `/Users/<your-username>/jenkins` or `/home/<your-username>/jenkins`. Note that if you change the source volume or directory for this, the volume from the `docker:dind` container above needs to be updated to match this.

9 Maps the `/certs/client` directory to the previously created `jenkins-docker-certs` volume. This makes the client TLS certificates needed to connect to the Docker daemon available in the path specified by the `DOCKER_CERT_PATH` environment variable.

10 Maps the `$HOME` directory on the host (i.e. your local) machine (usually the `/Users/<your-username>` directory) to the `/home` directory in the container.

11 The name of the Docker image, which you built in the previous step.

Note: If copying and pasting the command snippet above does not work, try copying and pasting this annotation-free version here:

```
docker run --name jenkins-blueocean --rm --detach \
  --network jenkins --env DOCKER_HOST=tcp://docker:2376 \
  --env DOCKER_CERT_PATH=/certs/client --env DOCKER_TLS_VERIFY=1 \
  --publish 8080:8080 --publish 50000:50000 \
  --volume jenkins-data:/var/jenkins_home \
  --volume jenkins-docker-certs:/certs/client:ro \
  --volume "$HOME":/home \
  myjenkins-blueocean:1.1
```

6. Proceed to the Post-installation setup wizard.

On Windows

The Jenkins project provides a Linux container image, not a Windows container image. Be sure that your Docker for Windows installation is configured to run `Linux Containers` rather than `Windows Containers`. See the Docker documentation for instructions to switch to Linux containers. Once configured to run `Linux Containers`, the steps are:

1. Open up a command prompt window and similar to the macOS and Linux instructions above do the following:
2. Create a bridge network in Docker

   ```
   docker network create jenkins
   ```

3. Run a docker:dind Docker image

   ```
   docker run --name jenkins-docker --rm --detach ^
     --privileged --network jenkins --network-alias docker ^
     --env DOCKER_TLS_CERTDIR=/certs ^
     --volume jenkins-docker-certs:/certs/client ^
     --volume jenkins-data:/var/jenkins_home ^
     docker:dind
   ```

4. Build a customised official Jenkins Docker image using above Dockerfile and `docker build` command.
5. Run your own `myjenkins-blueocean:1.1` image as a container in Docker using the following `docker run` command:

   ```
   docker run --name jenkins-blueocean --rm --detach ^
     --network jenkins --env DOCKER_HOST=tcp://docker:2376 ^
     --env DOCKER_CERT_PATH=/certs/client --env DOCKER_TLS_VERIFY=1 ^
     --volume jenkins-data:/var/jenkins_home ^
     --volume jenkins-docker-certs:/certs/client:ro ^
     --volume "%HOMEDRIVE%%HOMEPATH%":/home ^
     --publish 8080:8080 --publish 50000:50000 myjenkins-blueocean:1.1
   ```

6. Proceed to the Setup wizard.

Accessing the Docker container

If you have some experience with Docker and you wish or need to access your Docker container through a terminal/command prompt using the `docker exec` command, you can add an option like `--name jenkins-tutorial` to the `docker exec` command. That will access the Jenkins Docker container named "jenkins-tutorial".

This means you could access your docker container (through a separate terminal/command prompt window) with a `docker exec` command like:

```
docker exec -it jenkins-blueocean bash
```

Accessing the Docker logs

There is a possibility you may need to access the Jenkins console log, for instance, when Unlocking Jenkins as part of the Post-installation setup wizard.

The Jenkins console log is easily accessible through the terminal/command prompt window from which you executed the `docker run …` command. In case if needed you can also access the Jenkins console log through the Docker logs of your container using the following command:

```
docker logs <docker-container-name>
```

Your `<docker-container-name>` can be obtained using the `docker ps` command.

Accessing the Jenkins home directory

There is a possibility you may need to access the Jenkins home directory, for instance, to check the details of a Jenkins build in the `workspace` subdirectory.

If you mapped the Jenkins home directory (`/var/jenkins_home`) to one on your machine's local file system (i.e. in the `docker run …` command above), then you can access the contents of this directory through your machine's usual terminal/command prompt.

Otherwise, if you specified the `--volume jenkins-data:/var/jenkins_home` option in the `docker run …` command, you can access the contents of the Jenkins home directory through your container's terminal/command prompt using the `docker container exec` command:

```
docker container exec -it <docker-container-name> bash
```

As mentioned above, your `<docker-container-name>` can be obtained using the `docker container ls` command. If you specified the `--name jenkins-blueocean` option in the `docker container run …` command above (see also Accessing the Jenkins/Blue Ocean Docker container), you can simply use the `docker container exec` command:

```
docker container exec -it jenkins-blueocean bash
```

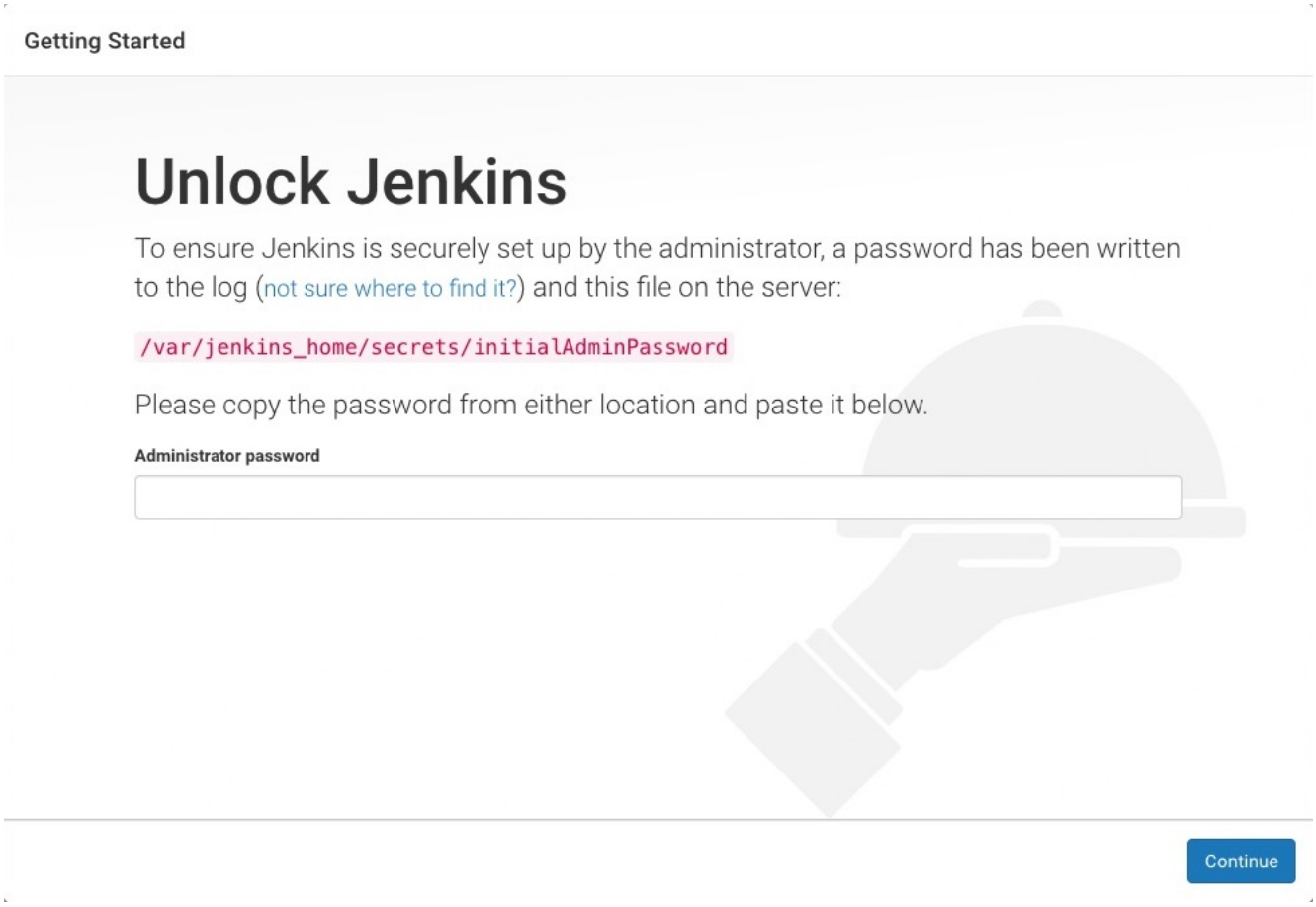
Setup wizard

Before you can access Jenkins, there are a few quick "one-off" steps you'll need to perform.

Unlocking Jenkins

When you first access a new Jenkins instance, you are asked to unlock it using an automatically-generated password.

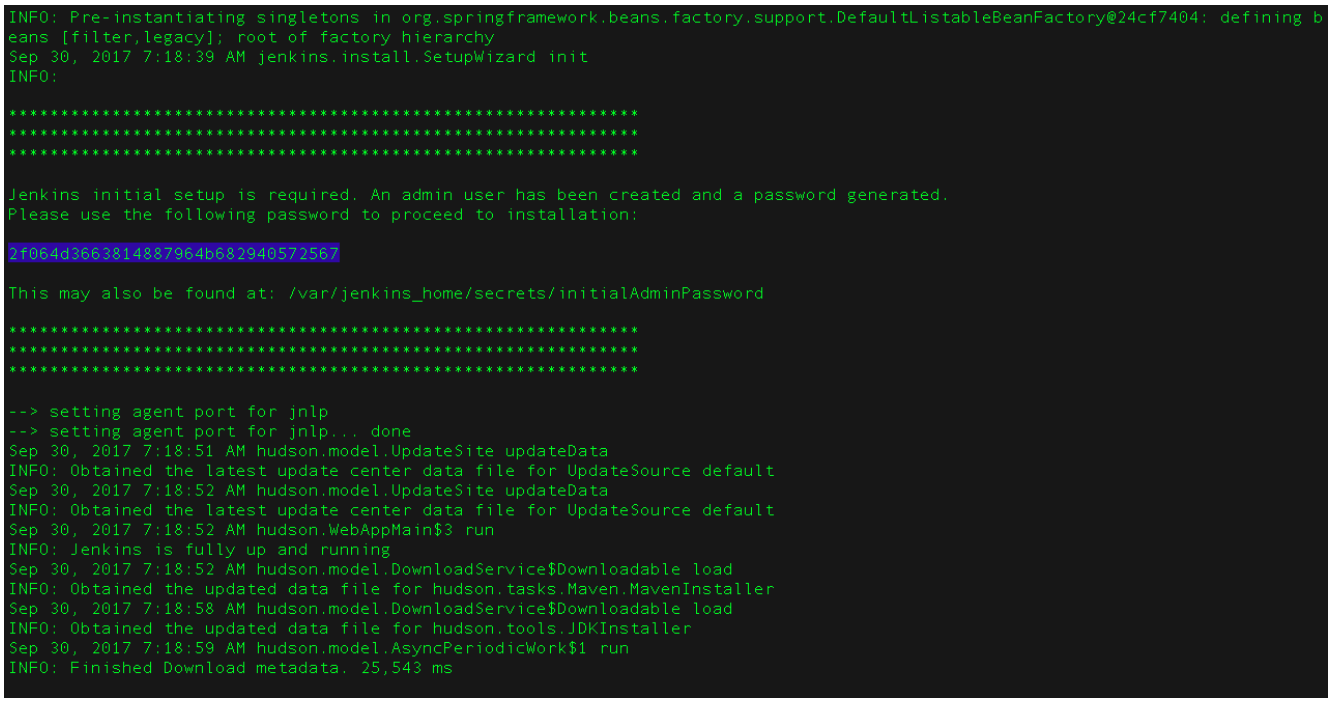
1. After the 2 sets of asterisks appear in the terminal/command prompt window, browse to `http://localhost:8080` and wait until the **Unlock Jenkins** page appears.



2. Display the Jenkins console log with the command:

   ```
   docker logs jenkins-blueocean
   ```

3. From your terminal/command prompt window again, copy the automatically-generated alphanumeric password (between the 2 sets of asterisks).



4. On the **Unlock Jenkins** page, paste this password into the **Administrator password** field and click **Continue**.

Customizing Jenkins with plugins

After unlocking Jenkins, the **Customize Jenkins** page appears.

On this page, click **Install suggested plugins**.

The setup wizard shows the progression of Jenkins being configured and the suggested plugins being installed. This process may take a few minutes.

Creating the first administrator user

Finally, Jenkins asks you to create your first administrator user.

1. When the **Create First Admin User** page appears, specify your details in the respective fields and click **Save and Finish**.
2. When the **Jenkins is ready** page appears, click **Start using Jenkins**.
   Notes:
   - This page may indicate **Jenkins is almost ready!** instead and if so, click **Restart**.
   - If the page doesn't automatically refresh after a minute, use your web browser to refresh the page manually.
3. If required, log in to Jenkins with the credentials of the user you just created and you're ready to start using Jenkins!

Stopping and restarting Jenkins

Throughout the remainder of this tutorial, you can stop your Docker container by running:

```
docker stop jenkins-blueocean jenkins-docker
```

To restart your Docker container:

1. Run the same `docker run …` commands you ran for macOS, Linux or Windows above.
2. Browse to `http://localhost:8080`.
3. Wait until the log in page appears and log in.

Fork and clone the sample repository

Obtain the simple "Hello world!" Java application from GitHub, by forking the sample repository of the application's source code into your own GitHub account and then cloning this fork locally.

1. Ensure you are signed in to your GitHub account. If you don't yet have a GitHub account, sign up for a free one on the GitHub website.
2. Fork the `simple-java-maven-app` on GitHub into your local GitHub account. If you need help with this process, refer to the Fork A Repo documentation on the GitHub website for more information.
3. Clone your forked `simple-java-maven-app` repository (on GitHub) locally to your machine. To begin this process, do either of the following (where `<your-username>` is the name of your user account on your operating system):
   - If you have the GitHub Desktop app installed on your machine:
     a. In GitHub, click the green **Clone or download** button on your forked repository, then **Open in Desktop**.

b. In GitHub Desktop, before clicking **Clone** on the **Clone a Repository** dialog box, ensure **Local Path** for:

- macOS is `/Users/<your-username>/Documents/GitHub/simple-java-maven-app`
- Linux is `/home/<your-username>/GitHub/simple-java-maven-app`
- Windows is `C:\Users\<your-username>\Documents\GitHub\simple-java-maven-app`

- Otherwise:

a. Open up a terminal/command line prompt and `cd` to the appropriate directory on:

- macOS - `/Users/<your-username>/Documents/GitHub/`
- Linux - `/home/<your-username>/GitHub/`
- Windows - `C:\Users\<your-username>\Documents\GitHub\` (although use a Git bash command line window as opposed to the usual Microsoft command prompt)

b. Run the following command to continue/complete cloning your forked repo:
`git clone https://github.com/YOUR-GITHUB-ACCOUNT-NAME/simple-java-maven-app`
where `YOUR-GITHUB-ACCOUNT-NAME` is the name of your GitHub account.

Create your Pipeline project in Jenkins

1. Go back to Jenkins, log in again if necessary and click **create new jobs** under **Welcome to Jenkins! Note:** If you don't see this, click **New Item** at the top left.
2. In the **Enter an item name** field, specify the name for your new Pipeline project (e.g. `simple-java-maven-app`).
3. Scroll down and click **Pipeline**, then click **OK** at the end of the page.
4. (*Optional*) On the next page, specify a brief description for your Pipeline in the **Description** field (e.g. `An entry-level Pipeline demonstrating how to use Jenkins to build a simple Java application with Maven.`)
5. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.
6. From the **Definition** field, choose the **Pipeline script from SCM** option. This option instructs Jenkins to obtain your Pipeline from Source Control Management (SCM), which will be your locally cloned Git repository.
7. From the **SCM** field, choose **Git**.
8. In the **Repository URL** field, specify the directory path of your locally cloned repository above, which is from your user account/home directory on your host machine, mapped to the `/home` directory of the Jenkins container - i.e.
   - For macOS - `/home/Documents/GitHub/simple-java-maven-app`
   - For Linux - `/home/GitHub/simple-java-maven-app`
   - For Windows - `/home/Documents/GitHub/simple-java-maven-app`

9. Click **Save** to save your new Pipeline project. You're now ready to begin creating your `Jenkinsfile`, which you'll be checking into your locally cloned Git repository.

Create your initial Pipeline as a Jenkinsfile

You're now ready to create your Pipeline that will automate building your Java application with Maven in Jenkins. Your Pipeline will be created as a `Jenkinsfile`, which will be committed to your locally cloned Git repository (`simple-java-maven-app`).

This is the foundation of "Pipeline-as-Code", which treats the continuous delivery pipeline as a part of the application to be versioned and reviewed like any other code. Read more about Pipeline and what a Jenkinsfile is in the Pipeline and Using a Jenkinsfile sections of the User Handbook.

First, create an initial Pipeline to download a Maven Docker image and run it as a Docker container (which will build your simple Java application). Also add a "Build" stage to the Pipeline that begins orchestrating this whole process.

1. Using your favorite text editor or IDE, create and save new text file with the name `Jenkinsfile` at the root of your local `simple-java-maven-app` Git repository.

2. Copy the following Declarative Pipeline code and paste it into your empty `Jenkinsfile`:

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine' (1)
            args '-v /root/.m2:/root/.m2' (2)
        }
    }
    stages {
        stage('Build') { (3)
            steps {
                sh 'mvn -B -DskipTests clean package' (4)
            }
        }
    }
}
```

1 This `image` parameter (of the `agent` section's `docker` parameter) downloads the `maven:3-alpine` Docker image (if it's not already available on your machine) and runs this image as a separate container. This means that:

- You'll have separate Jenkins and Maven containers running locally in Docker.
- The Maven container becomes the agent that Jenkins uses to run your Pipeline project. However, this container is short-lived - its lifespan is only that of the duration of your Pipeline's execution.

2 This `args` parameter creates a reciprocal mapping between the `/root/.m2` (i.e. Maven repository) directories in the short-lived Maven Docker container and that of your Docker host's filesystem. Explaining the details behind this is beyond the scope of this tutorial. However, the main reason for doing this is to ensure that the artifacts necessary to build your Java application (which Maven downloads while your Pipeline is being executed) are retained in the Maven repository beyond the lifespan of the Maven container. This prevents Maven from having to download the same artifacts during successive runs of your Jenkins Pipeline, which you'll be conducting later on. Be aware that unlike the Docker data volume you created for `jenkins-data` above, the Docker host's filesystem is effectively cleared out each time Docker is restarted. This means you'll lose the downloaded Maven repository artifacts each time Docker restarts.

3 Defines a `stage` (directive) called `Build` that appears on the Jenkins UI.

4 This `sh` step (of the `steps` section) runs the Maven command to cleanly build your Java application

(without running any tests).

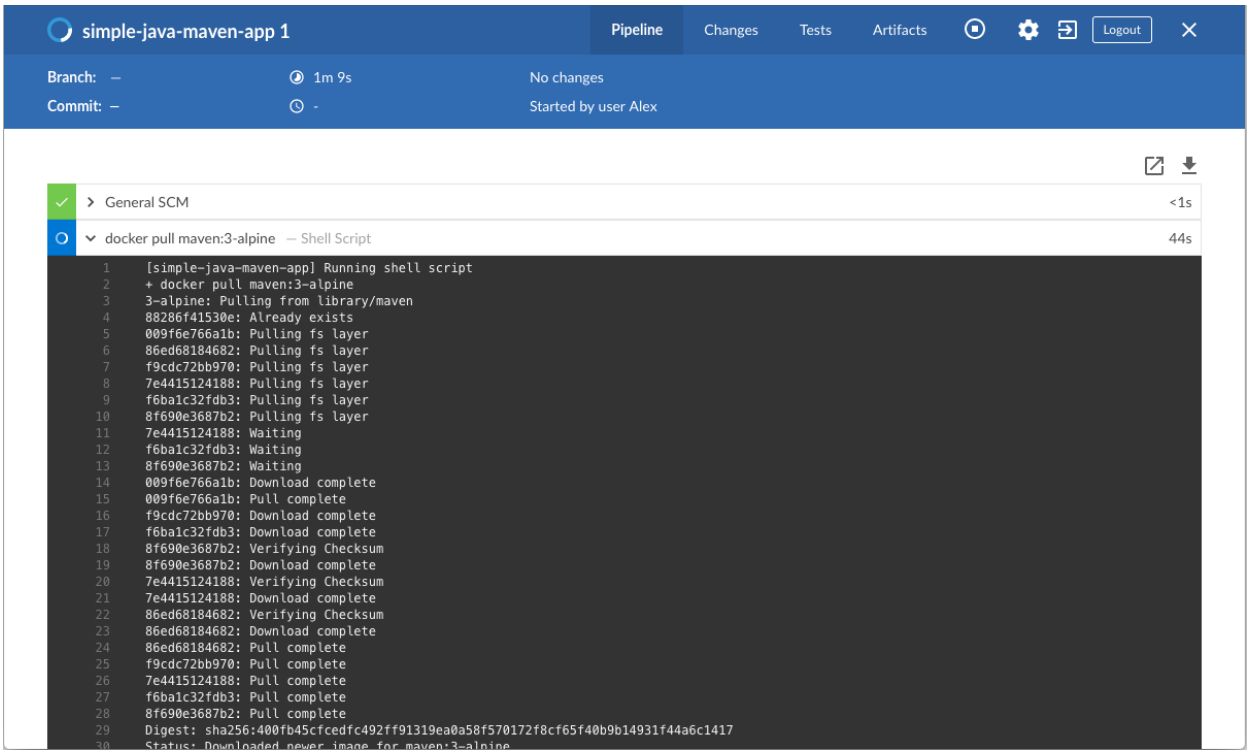
3. Save your edited `Jenkinsfile` and commit it to your local `simple-java-maven-app` Git repository. E.g. Within the `simple-java-maven-app` directory, run the commands:
   `git add .`
   then
   `git commit -m "Add initial Jenkinsfile"`

4. Go back to Jenkins again, log in again if necessary and click **Open Blue Ocean** on the left to access Jenkins's Blue Ocean interface.

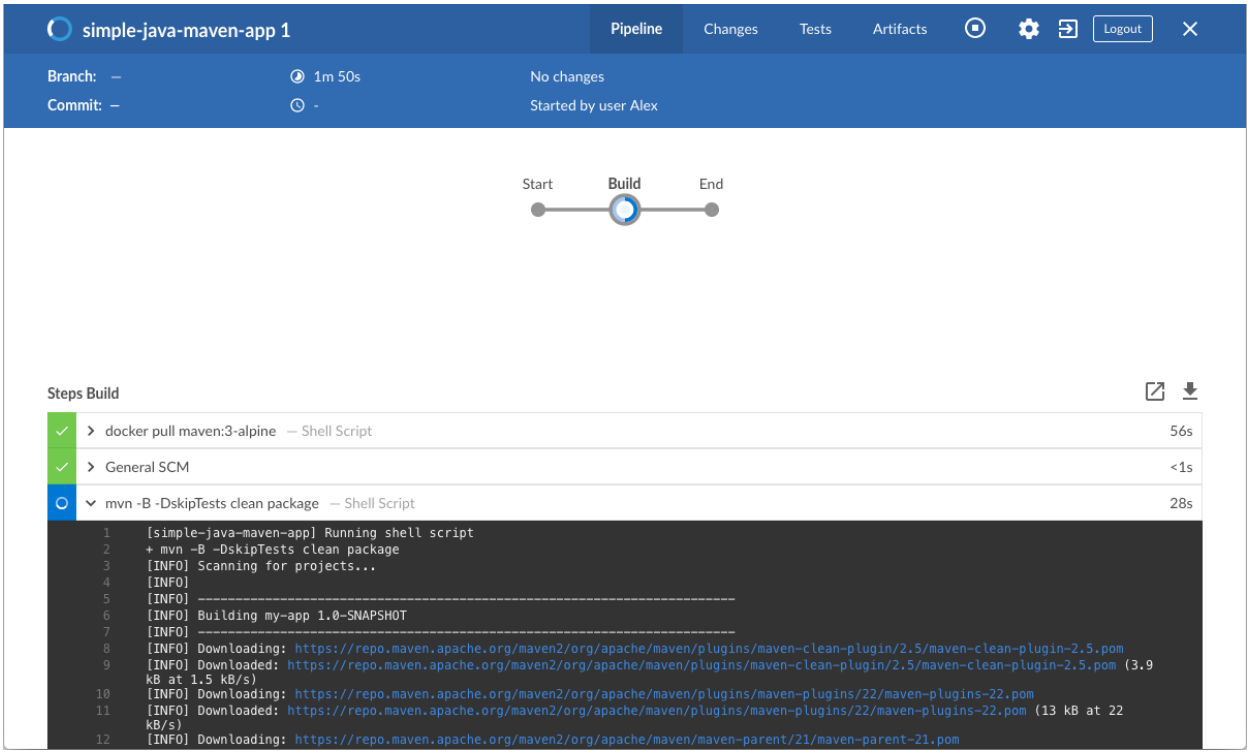
5. In the **This job has not been run** message box, click **Run**, then quickly click the **OPEN** link which appears briefly at the lower-right to see Jenkins running your Pipeline project. If you weren't able to click the **OPEN** link, click the row on the main Blue Ocean interface to access this feature.
   Note: You may need to wait several minutes for this first run to complete. After making a clone of your local `simple-java-maven-app` Git repository itself, Jenkins:

   a. Initially queues the project to be run on the agent.

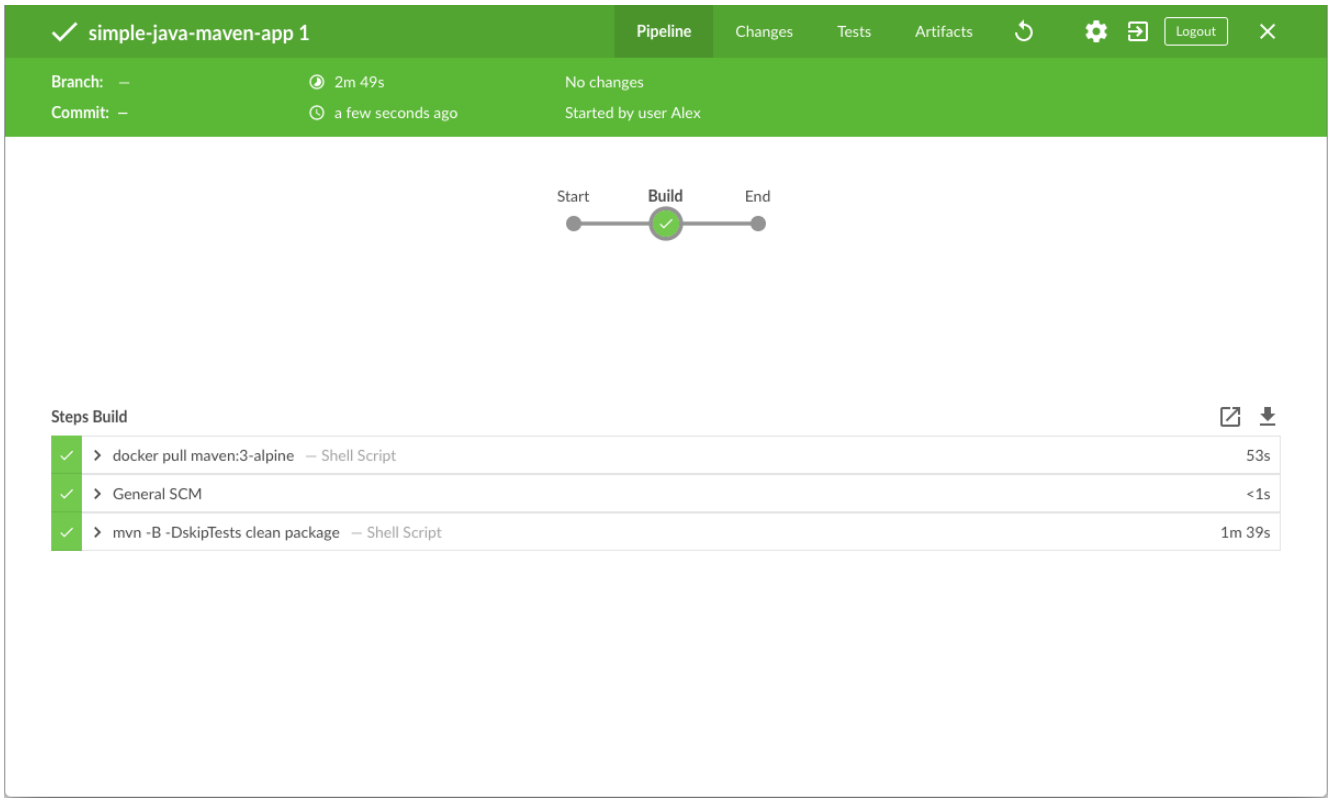
   b. Downloads the Maven Docker image and runs it in a container on Docker.



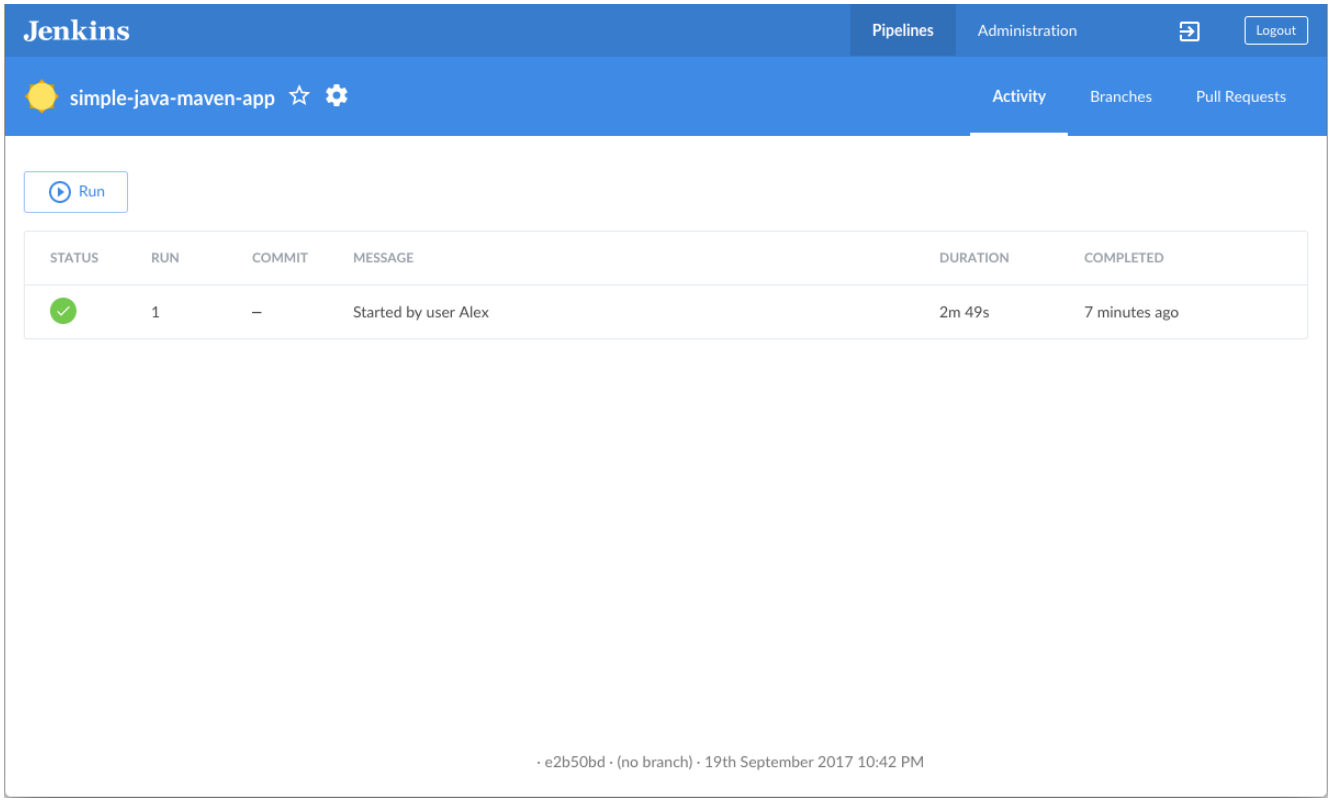
   c. Runs the `Build` stage (defined in the `Jenkinsfile`) on the Maven container. During this time, Maven downloads many artifacts necessary to build your Java application, which will ultimately be stored in Jenkins's local Maven repository (in the Docker host's filesystem).



The Blue Ocean interface turns green if Jenkins built your Java application successfully.



6. Click the **X** at the top-right to return to the main Blue Ocean interface.



Add a test stage to your Pipeline

1. Go back to your text editor/IDE and ensure your `Jenkinsfile` is open.
2. Copy and paste the following Declarative Pipeline syntax immediately under the `Build` stage of your `Jenkinsfile`:

```
        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }
```

so that you end up with:

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Test') { (1)
            steps {
                sh 'mvn test' (2)
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml' (3)
```

```
                }
            }
        }
    }
}
```

1 Defines a `stage` (directive) called `Test` that appears on the Jenkins UI.

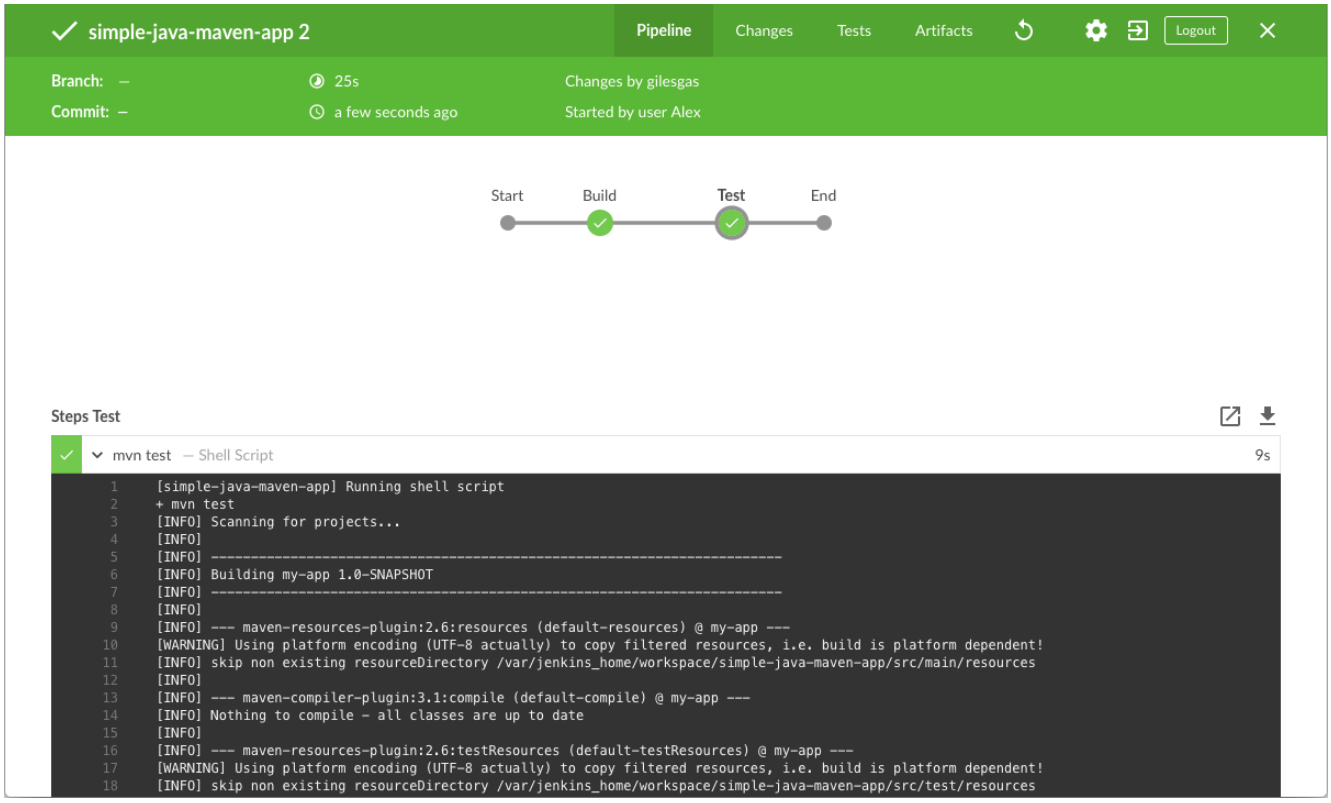
2 This `sh` step (of the `steps` section) executes the Maven command to run the unit test on your simple Java application. This command also generates a JUnit XML report, which is saved to the `target/surefire-reports` directory (within the `/var/jenkins_home/workspace/simple-java-maven-app` directory in the Jenkins container).

3 This `junit` step (provided by the JUnit Plugin) archives the JUnit XML report (generated by the `mvn test` command above) and exposes the results through the Jenkins interface. In Blue Ocean, the results are accessible through the **Tests** page of a Pipeline run. The `post` section's `always` condition that contains this `junit` step ensures that the step is *always* executed *at the completion* of the `Test` stage, regardless of the stage's outcome.

3. Save your edited `Jenkinsfile` and commit it to your local `simple-java-maven-app` Git repository. E.g. Within the `simple-java-maven-app` directory, run the commands:
   `git stage .`
   then
   `git commit -m "Add 'Test' stage"`

4. Go back to Jenkins again, log in again if necessary and ensure you've accessed Jenkins's Blue Ocean interface.

5. Click **Run** at the top left, then quickly click the **OPEN** link which appears briefly at the lower-right to see Jenkins running your amended Pipeline project. If you weren't able to click the **OPEN** link, click the *top* row on the Blue Ocean interface to access this feature.
   Note: You'll notice from this run that Jenkins no longer needs to download the Maven Docker image. Instead, Jenkins only needs to run a new container from the Maven image downloaded previously. Also, if Docker had not restarted since you last ran the Pipeline above, then no Maven artifacts need to be downloaded during the "Build" stage. Therefore, running your Pipeline this subsequent time should be much faster.
   If your amended Pipeline ran successfully, here's what the Blue Ocean interface should look like. Notice the additional "Test" stage. You can click on the previous "Build" stage circle to access the output from that stage.



6. Click the **X** at the top-right to return to the main Blue Ocean interface.

Add a final deliver stage to your Pipeline

1. Go back to your text editor/IDE and ensure your `Jenkinsfile` is open.
2. Copy and paste the following Declarative Pipeline syntax immediately under the `Test` stage of your `Jenkinsfile`:

```
        stage('Deliver') {
            steps {
                sh './jenkins/scripts/deliver.sh'
            }
        }
```

and add a `skipStagesAfterUnstable` option so that you end up with:

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
```

```
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }
        stage('Deliver') { (1)
            steps {
                sh './jenkins/scripts/deliver.sh' (2)
            }
        }
    }
}
```

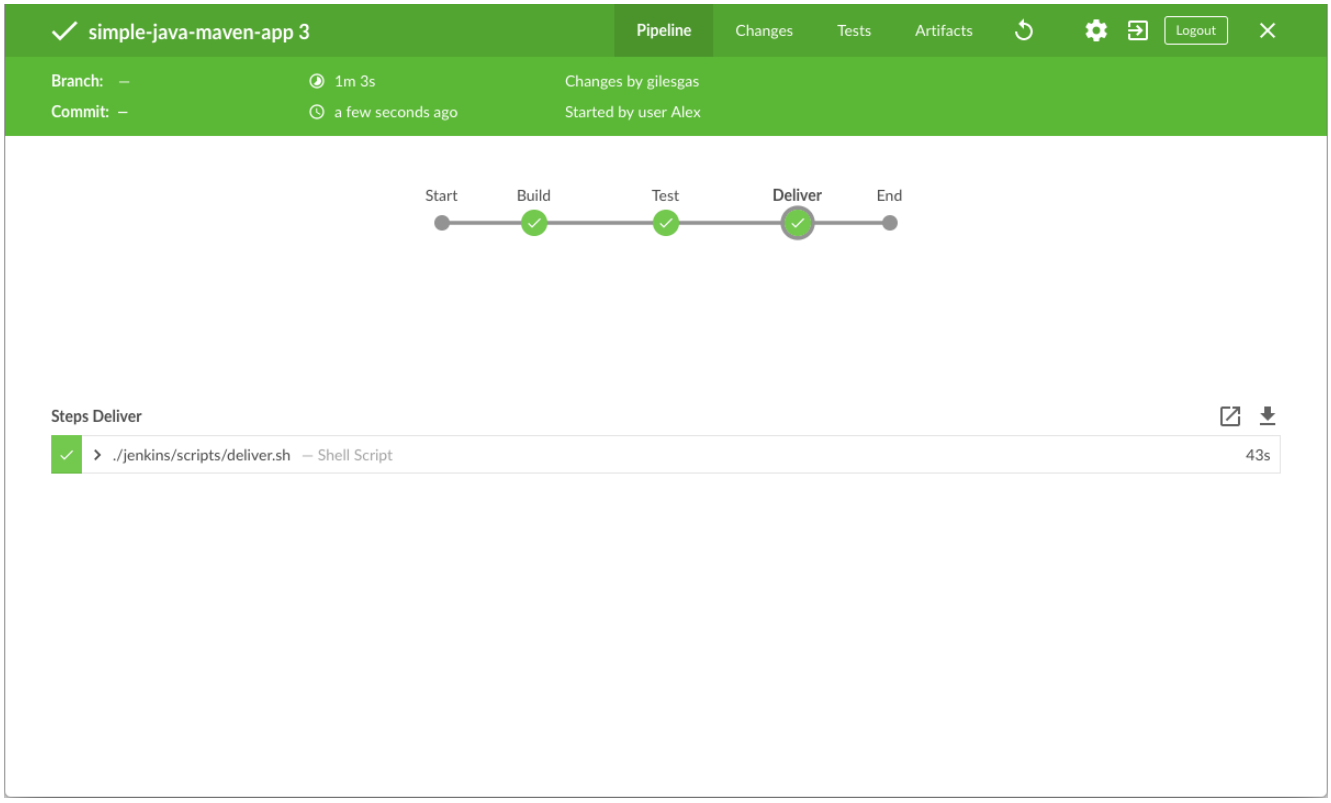
1 Defines a new stage called `Deliver` that appears on the Jenkins UI.

2 This `sh` step (of the `steps` section) runs the shell script `deliver.sh` located in the `jenkins/scripts` directory from the root of the `simple-java-maven-app` repository. Explanations about what this script does are covered in the `deliver.sh` file itself. As a general principle, it's a good idea to keep your Pipeline code (i.e. the `Jenkinsfile`) as tidy as possible and place more complex build steps (particularly for stages consisting of 2 or more steps) into separate shell script files like the `deliver.sh` file. This ultimately makes maintaining your Pipeline code easier, especially if your Pipeline gains more complexity.

3. Save your edited `Jenkinsfile` and commit it to your local `simple-java-maven-app` Git repository. E.g. Within the `simple-java-maven-app` directory, run the commands:
   `git stage .`
   then
   `git commit -m "Add 'Deliver' stage"`

4. Go back to Jenkins again, log in again if necessary and ensure you've accessed Jenkins's Blue Ocean interface.

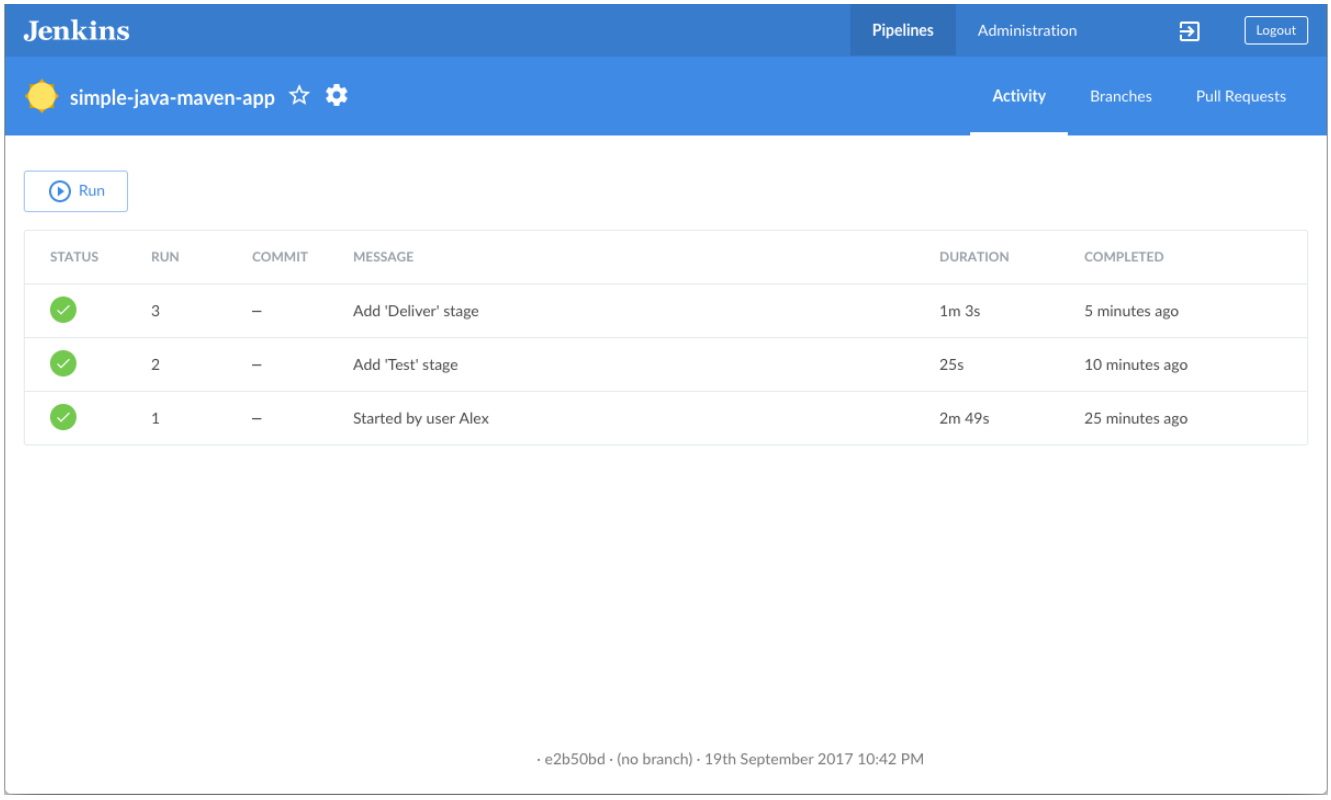
5. Click **Run** at the top left, then quickly click the **OPEN** link which appears briefly at the lower-right to see Jenkins running your amended Pipeline project. If you weren't able to click the **OPEN** link, click the *top* row on the Blue Ocean interface to access this feature.
   If your amended Pipeline ran successfully, here's what the Blue Ocean interface should look like. Notice the additional "Deliver" stage. Click on the previous "Test" and "Build" stage circles to access the outputs from those stages.



Here’s what the output of the "Deliver" stage should look like, showing you the execution results of your Java application at the end.

```
[INFO] ------------------------------------------------------------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-cli) @ my-app ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-cli) @ my-app ---
[INFO] Installing /var/jenkins_home/workspace/simple-java-maven-app/target/my-app-1.0-SNAPSHOT.jar to
/root/.m2/repository/com/mycompany/app/my-app/1.0-SNAPSHOT/my-app-1.0-SNAPSHOT.jar
[INFO] Installing /var/jenkins_home/workspace/simple-java-maven-app/pom.xml to /root/.m2/repository/com/mycompany/app/my-app/1.0-
SNAPSHOT/my-app-1.0-SNAPSHOT.pom
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-help-plugin:2.2:evaluate (default-cli) @ my-app ---
[INFO] No artifact parameter specified, using 'com.mycompany.app:my-app:jar:1.0-SNAPSHOT' as project.
[INFO]
my-app
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.870 s
[INFO] Finished at: 2017-10-01T13:20:22Z
[INFO] Final Memory: 18M/96M
[INFO] ------------------------------------------------------------------------
+ set +x
The following complex command extracts the value of the <name/> element
within <project/> of your Java/Maven projects "pom.xml" file.
++ mvn help:evaluate -Dexpression=project.name
++ grep '^[^\[]'
+ NAME=my-app
+ set +x
The following complex command behaves similarly to the previous one but
extracts the value of the <version/> element within <project/> instead.
++ mvn help:evaluate -Dexpression=project.version
++ grep '^[^\[]'
+ VERSION=1.0-SNAPSHOT
+ set +x
The following command runs and outputs the execution of your Java
application (which Jenkins built using Maven) to the Jenkins UI.
+ java -jar target/my-app-1.0-SNAPSHOT.jar
Hello World!
```

6. Click the **X** at the top-right to return to the main Blue Ocean interface, which lists your previous Pipeline runs in reverse chronological order.



Wrapping up

Well done! You’ve just used Jenkins to build a simple Java application with Maven!

The "Build", "Test" and "Deliver" stages you created above are the basis for building more complex Java applications with Maven in Jenkins, as well as Java and Maven applications that integrate with other technology stacks.

Because Jenkins is extremely extensible, it can be modified and configured to handle practically any aspect of build orchestration and automation.

To learn more about what Jenkins can do, check out:

- The Tutorials overview page for other introductory tutorials.
- The User Handbook for more detailed information about using Jenkins, such as Pipelines (in particular Pipeline syntax) and the Blue Ocean interface.
- The Jenkins blog for the latest events, other tutorials and updates.

- Events
- Mailing lists
- Chats
- Special Interest Groups
- Twitter
- Reddit

Other

- Code of Conduct
- Press information
- Merchandise
- Artwork
- Awards