# Setting up a test job

To test that your Jenkins server and slaves are running correctly and able to run Docker containers we will set up a basic job to test the base functionality

First from the Jenkins home page click new item

Select freestyle project and give it an appropriate name (I used test-job)

This job will not require much configuration so on the next page scroll to build click add build step and execute shell

Paste the following into the command box

*docker -v*

*docker run ubuntu /bin/echo hello world*

*docker stop $(docker ps -a -q)*

*docker rm $(docker ps -a -q)*

*docker rmi $(docker images -q)*

this shell script will print to the console the version of docker your running then start a basic Ubuntu image which prints hello world then stops all running containers, removes them and finally removes all the docker images on the slave (do not use this once you have more jobs set up as you will not want to be removing all the containers encase they are being used by other tests)

next click to save the job and click build now to test your setup

if you want you can click the arrow next to the running job and click console output to see live what is happening on the slave and check that it is running as expected (you will need to do this once the job has finished running anyway)

once the job has finished running and you are viewing the console output check that you can see in the logs the docker version, the hello world command and finally the stopping and clean-up of containers

if this all ran as expected then you have proven that you are able to pull and run docker images on your slave

once you have this job passing it is recommended to make some modifications to it to allow you to run it to test your slaves without interfering with other jobs you can do this by modifying the shell command and giving the container a name then only removing the named container and the specific image that you created an example of this is below

*docker -v*

*docker run -d --name testUbuntu ubuntu /bin/echo hello worlddocker stop $(docker ps -a -q)*

*docker logs testUbuntu*

*dockrr stop testUbuntu*

*docker rm testUbuntu*

*docker rmi Ubuntu*

if you make these changes make sure to rerun the job to ensure that is still passes

# Pulling from a git repo

To create a project based on a git repo it is similar to the test job that we set up, first create an appropriately named freestyle job but this time under source code management select git (if you cannot see git in the list of providers you will want to check that the git plugin has been successfully installed and resolve any dependency errors that it might have)

Next input the url of the git project you wish to clone. If your project requires you to be logged in to download it then you create a credentials object using the dropdown below the url box. And specifying the details of the user you want Jenkins to use (it is recommended that you create Jenkins its own user for this purpose)

In the next box, you can specify what branch you want to check out when the git repository is cloned

To check this is able to pull down the repo correctly just add a shell script to the build commands that’s runs the command ‘ls’ this way when you run the job you can check in the console that you see the files for the repo that you expected

If this worked as expected, then you can now modify the shell script to build your project.

The rest of this guide will be focusing on building, deploying and testing Docker images

# Building a Docker image

(recommended to install the plugin [PostBuildScript](https://wiki.jenkins-ci.org/display/JENKINS/PostBuildScript+Plugin))

Continuing on from the git project all we will need to do to build a docker image is change the build script from ls to

*docker build -t <what you want to name your image> .*

*docker run -d -t –name <what you want to call your container> <the name of your image>*

you can add any needed environment variables to the docker run command with the command -e e.g -e foo=bar -e foo1=bar1

if you ran this job as it is and your docker image is valid it would run correctly the first time but fail on future runs. This is because you build an image and then run a container from it but never clean up after the build is finished so the next time the image tries to build it will conflict with the previous image.

To fix this we will need to add another shell script with the commands

*docker stop <your container name>*

*docker rm <your container name>*

*docker rmi <your image name>*

with these changes in place you should be able to build a working image as many times as you want as now after the build the images and containers are removed. However you will still have some problems if a build fails part way through as then the cleanup steps would never be reached. This is where the post build script plugin comes in handy

once the plugin is installed under post build actions select execute a set of scripts

this will add a new box where you can select add a build step and then add a shell script and copy the cleanup code into this

make sure Execute script only if build succeeds is not ticked and then your build should be ready to go

# Running unit tests

Running unit tests based on the previous job is similar to running them in a development environment simply add a new shell script after building and running a image that runs your unit tests, if they pass the build will succeed if they fail the build will fail.

Personally I try to ensure that I can run my unit tests from a make file with a command such as

make unittest

this way my Jenkins job simply has to run the command

docker exec <container name> make unittest

and doesn’t have to worry about what language the program is coded in

# Pushing a Docker image to repository

# Deploying a image to a rancher server

# Running database migrations

# Running acceptance tests

If your running acceptance tests in a Docker container the process is nearly identical to building a Docker image so follow the same steps as you did then just using your tests repo

Once you have this set up we have a couple of changes to make to see the output of the tests and run them slightly more efficiently

First in the first shell script remove -d from the Docker run command. -d runs the container detached meaning we cannot see the output of anything running inside it which we would want to do in the acceptance tests. The reason we do not want this in our other containers is because the acceptance tests run the terminate closing the container while the other containers stay running leaving us stuck monitoring logs but unable to continue.

Next in the cleanup script remove the Docker stop line. The acceptance tests exits the container once they have run so we don’t need to do this step

With these changes in place you should now be able to run your acceptance tests successfully

# Running accessibility tests

# Running performance tests