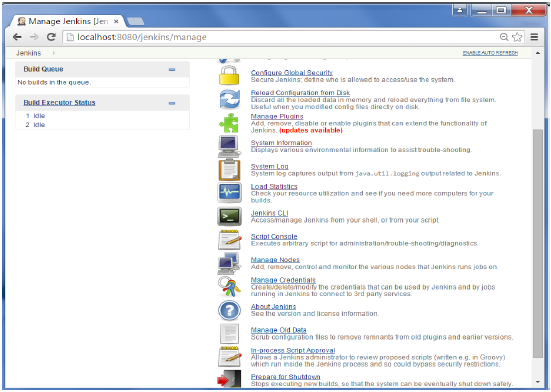
Distributed Builds

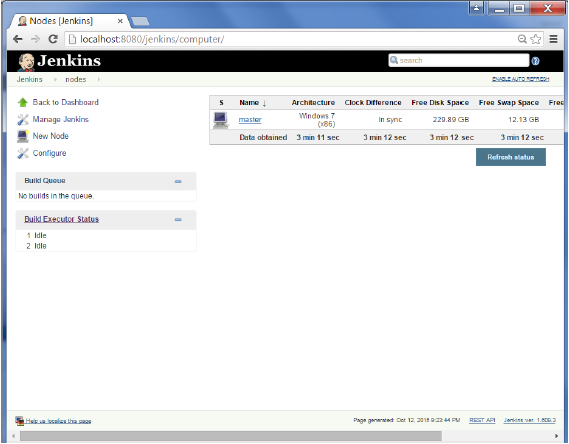
 slave is a computer that is set up to offload build projects from the master and once setup this distribution of tasks is fairly automatic. The exact delegation behavior depends on the configuration of each project; some projects may choose to "stick" to a particular machine for a build, while others may choose to roam freely between slaves.

Since each slave runs a separate program called a "slave agent" there is no need to install the full Jenkins (package or compiled binaries) on a slave. There are various ways to start slave agents, but in the end the slave agent and Jenkins master needs to establish a bi-directional communication link (for example a TCP/IP socket.) in order to operate.

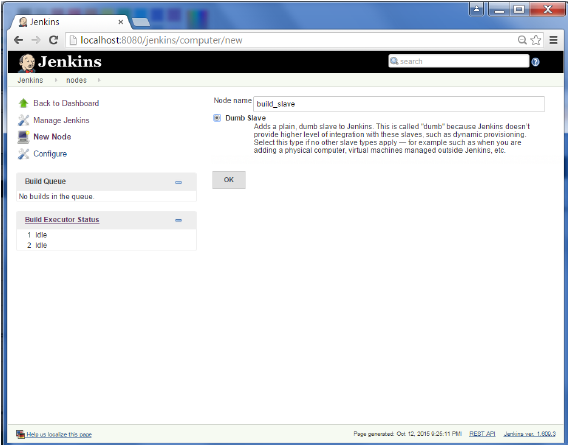
**Step 1** − Go to the Manage Jenkins section and scroll down to the section of Manage Nodes.



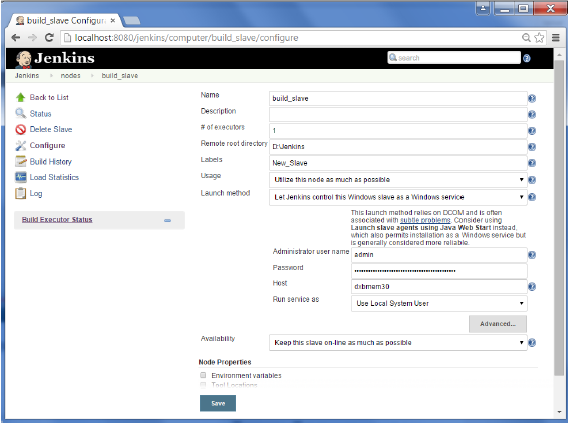
**Step 2** − Click on New Node



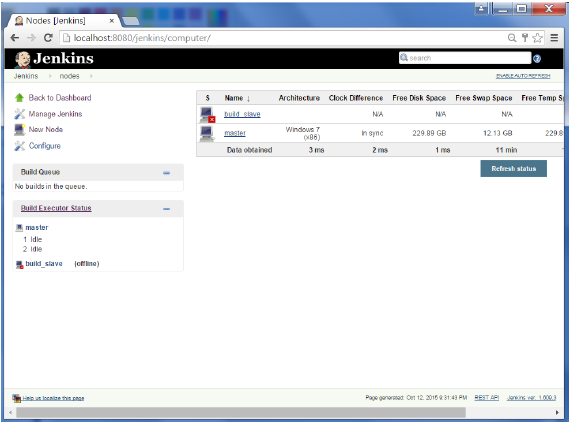
**Step 3** − Give a name for the node, choose the Dumb slave option and click on Ok.



**Step 4** − Enter the details of the node slave machine. In the below example, we are considering the slave machine to be a windows machine, hence the option of “Let Jenkins control this Windows slave as a Windows service” was chosen as the launch method. We also need to add the necessary details of the slave node such as the node name and the login credentials for the node machine. Click the Save button. The Labels for which the name is entered as “New\_Slave” is what can be used to configure jobs to use this slave machine.

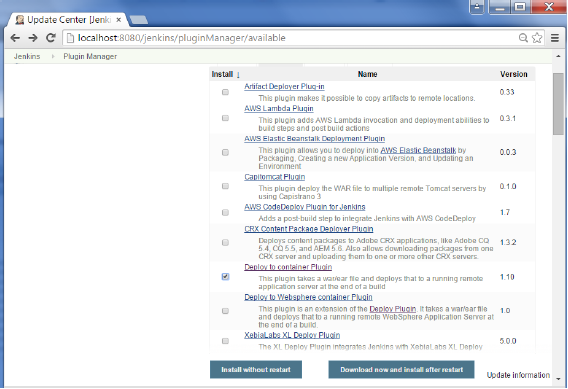


Once the above steps are completed, the new node machine will initially be in an offline state, but will come online if all the settings in the previous screen were entered correctly. One can at any time make the node slave machine as offline if required.



# Automated Deployment

**Step 1** − Go to Manage Jenkins → Manage Plugins. Go to the Available section and find the plugin “Deploy to container Plugin” and install the plugin. Restart the Jenkins server.



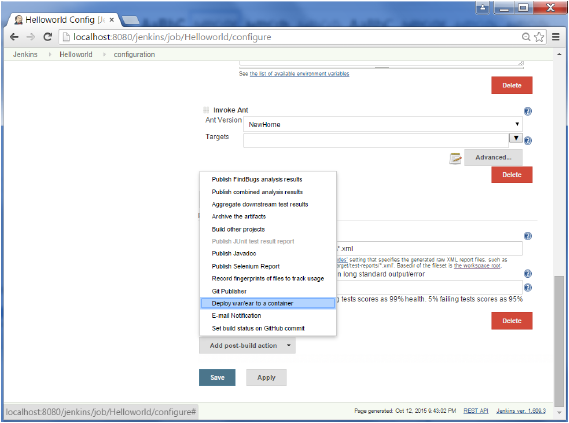
This plugin takes a war/ear file and deploys that to a running remote application server at the end of a build.

Tomcat 4.x/5.x/6.x/7.x

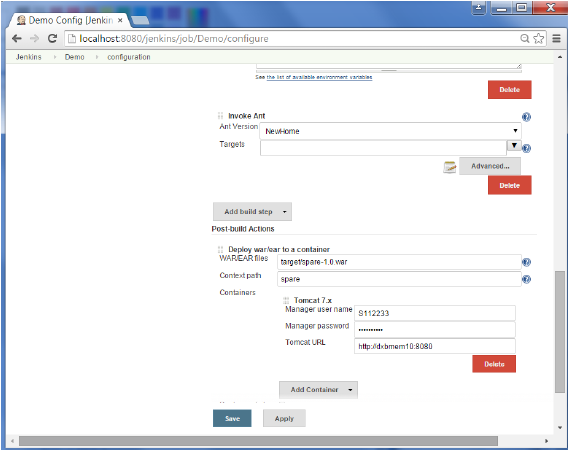
JBoss 3.x/4.x

Glassfish 2.x/3.x

**Step 2** − Go to your Build project and click the Configure option. Choose the option “Deploy war/ear to a container”



**Step 3** − In the Deploy war/ear to a container section, enter the required details of the server on which the files need to be deployed and click on the Save button. These steps will now ensure that the necessary files get deployed to the necessary container after a successful build.



## **Our application**

For this guide, we'll be using a very basic example: a Hello World server written with Node. Place this in a main.js:

// load the http module

var http = require('http');

// configure our HTTP server

var server = http.createServer(function (request, response) {

response.writeHead(200, {"Content-Type": "text/plain"});

response.end("Hello getintodevops.com\n");

});

// listen on localhost:8000

server.listen(8000);

console.log("Server listening at http://127.0.0.1:8000/");

We'll also need a package.json, which tells Node [some basic things](https://docs.npmjs.com/files/package.json) about our application:

{

"name": "getintodevops-hellonode",

"version": "1.0.0",

"description": "A Hello World HTTP server",

"main": "main.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"start": "node main.js"

},

"repository": {

"type": "git",

"url": "https://github.com/getintodevops/hellonode/"

},

"keywords": [

"node",

"docker",

"dockerfile"

],

"author": "miiro@getintodevops.com",

"license": "ISC"

}

**Pro tip:** Get all the code for this guide from <https://github.com/getintodevops/hellonode>

## **Writing a Dockerfile**

To be able to build a Docker image with our app, we'll need a Dockerfile. You can think of it as a blueprint for Docker: it tells Docker what the contents and parameters of our image should be.

Docker images are often based on other images. For this exercise, we are basing our image on the official Node Docker image. This makes our job easy, and our Dockerfile very short. The grunt work of installing Node and its dependencies in the image is already done in our base image; we'll just need to include our application.

The Dockerfile is best stored with the code - this way any changes to it are versioned along with the actual application code.

Add the following to a file called Dockerfile in the project directory:

# use a node base image

FROM node:7-onbuild

# set maintainer

LABEL maintainer "miiro@getintodevops.com"

# set a health check

HEALTHCHECK --interval=5s \

--timeout=5s \

CMD curl -f http://127.0.0.1:8000 || exit 1

# tell docker what port to expose

EXPOSE 8000

In short, the above Dockerfile instructs Docker to do the following:

* Use the node:7-onbuild image as the base for our image
* Set a label with the maintainer (not required, but good practice)
* Set a health check for the container (for Docker to be able to tell if the server is actually up or not)
* Tell Docker which port our server runs on

Additionally, our image inherits the following actions from the official node onbuild image:

* Copy all files in the current directory to /usr/src/app inside the image
* Run npm install to install any dependencies for app (if we had any)
* Specify npm start as the command Docker runs when the container starts

## **Building the image locally**

To build the image on your own computer, navigate to the project directory (the one with your application code and the Dockerfile), and run docker build:

docker build . -t getintodevops-hellonode:1

This instructs Docker to build the Dockerfile in the current directory with the tag getintodevops-hellonode:1. You will see Docker execute all the actions we specified in the Dockerfile (plus [the ones from the onbuild image](https://github.com/nodejs/docker-node/blob/1d328d2d967bd3a8d9b0260566383775d1a4aecc/7.7/onbuild/Dockerfile)).

### **RUNNING THE IMAGE LOCALLY**

If the above build command ran without errors, congratulations: your first Docker image is ready!

Let's make sure the image works as expected by running it:

docker run -it -p 8000:8000 getintodevops-hellonode:1

The above command tells Docker to run the image **i**nteractively with a pseudo-**t**ty, and map the port 8000 in the container to port 8000 in your machine.

You should now be able to check if the server responds in your local port 8000:

curl http://127.0.0.1:8000

Assuming it does, you can quit the docker run command with CTRL + C.

## **Building the image in Jenkins**

Now that we know our Docker image can be built, we'll want to do it automatically every time there is a change to the application code.

For this, we'll use Jenkins. Jenkins is an automation server often used to build and deploy applications.

Note: this guide assumes you are running Jenkins 2.0 or newer, with the Docker Pipeline plugin and Docker installed.

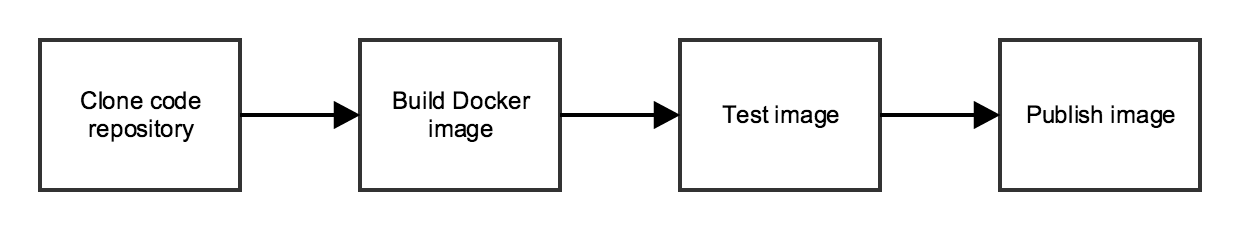
If you don't have access to a Jenkins installation, refer to <https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins>

### **PIPELINES AS CODE: THE JENKINSFILE**

Just like Dockerfiles, I'm a firm believer in storing Jenkins pipeline configuration as code, along with the application code.

It generally makes sense to have everything in the same repository; the application code, what the build artifact should look like (Dockerfile), and how said artifact is created automatically (Jenkinsfile).

Let's think about our pipeline for a second. We can identify four stages:



We'll need to tell Jenkins what our stages are, and what to do in each one of them. For this we'll write a Jenkins Pipeline specification in a Jenkinsfile.

Place the following into a file called Jenkinsfile in the same directory as the rest of the files:

node {

def app

stage('Clone repository') {

/\* Let's make sure we have the repository cloned to our workspace \*/

checkout scm

}

stage('Build image') {

/\* This builds the actual image; synonymous to

\* docker build on the command line \*/

app = docker.build("getintodevops/hellonode")

}

stage('Test image') {

/\* Ideally, we would run a test framework against our image.

\* For this example, we're using a Volkswagen-type approach ;-) \*/

app.inside {

sh 'echo "Tests passed"'

}

}

stage('Push image') {

/\* Finally, we'll push the image with two tags:

\* First, the incremental build number from Jenkins

\* Second, the 'latest' tag.

\* Pushing multiple tags is cheap, as all the layers are reused. \*/

docker.withRegistry('https://registry.hub.docker.com', 'docker-hub-credentials') {

app.push("${env.BUILD\_NUMBER}")

app.push("latest")

}

}

}

**Learn how to do this and much more with an instructor:**[**Get 86% OFF our DevOps Masterclass on Udemy for a limited time!**](https://www.udemy.com/devops-masterclass/?couponCode=GIDWEBARTICLEPROMO10)

That's the entirety of our pipeline specification for Jenkins. Now, we'll just need to tell Jenkins two things:

* Where to find our code
* What credentials to use to publish the Docker image

We'll start by configuring the credentials:

### **CONFIGURING DOCKER HUB WITH JENKINS**

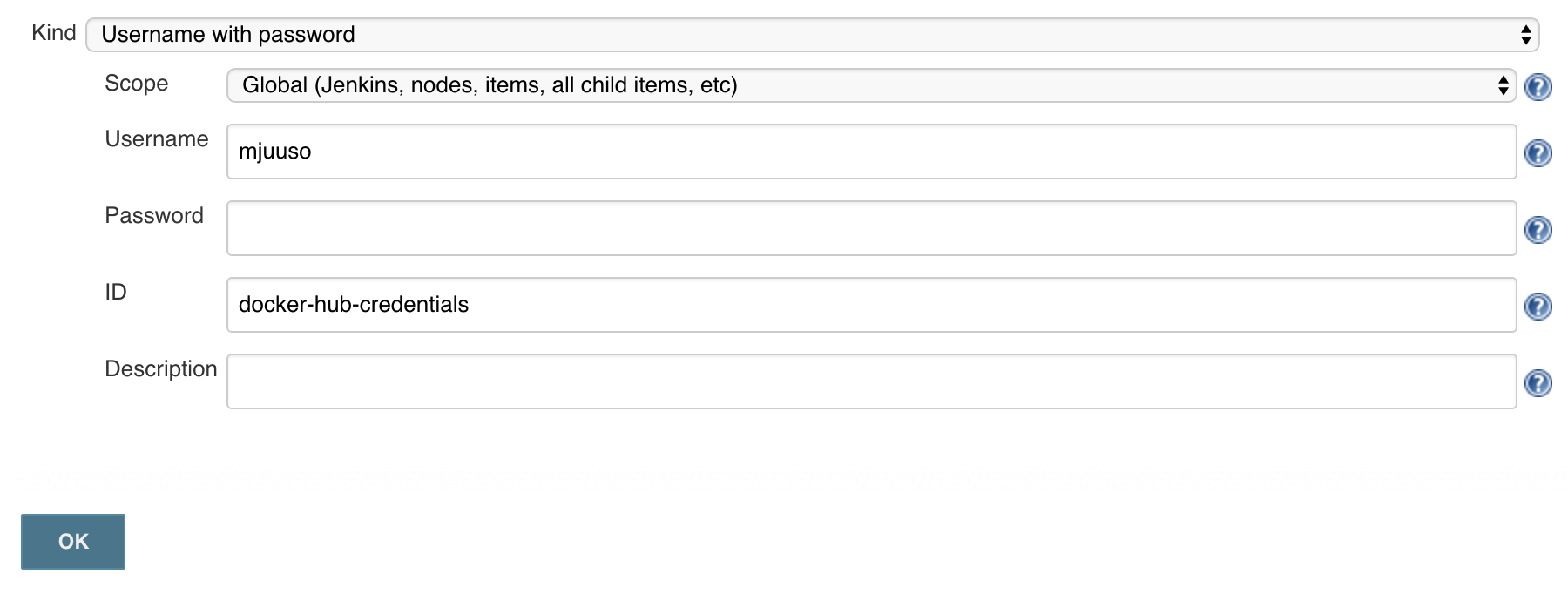
To store the Docker image resulting from our build, we'll be using Docker Hub. You can sign up for a free account at [https://hub.docker.com](https://hub.docker.com/).

We'll need to give Jenkins access to push the image to Docker Hub. For this, we'll create Credentials in Jenkins, and refer to them in the Jenkinsfile.

As you might have noticed in the above Jenkinsfile, we're using docker.withRegistry to wrap the app.push commands - this instructs Jenkins to log in to a specified registry with the specified credential id (docker-hub-credentials).



On the Jenkins front page, click on **Credentials** -> **System**-> **Global credentials -> Add Credentials**

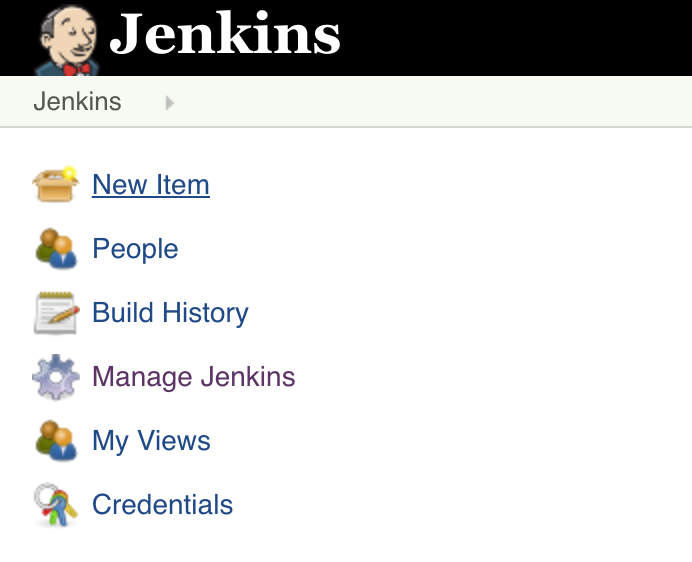


Add your Docker Hub credentials as the type **Username with password**, with the ID **docker-hub-credentials**

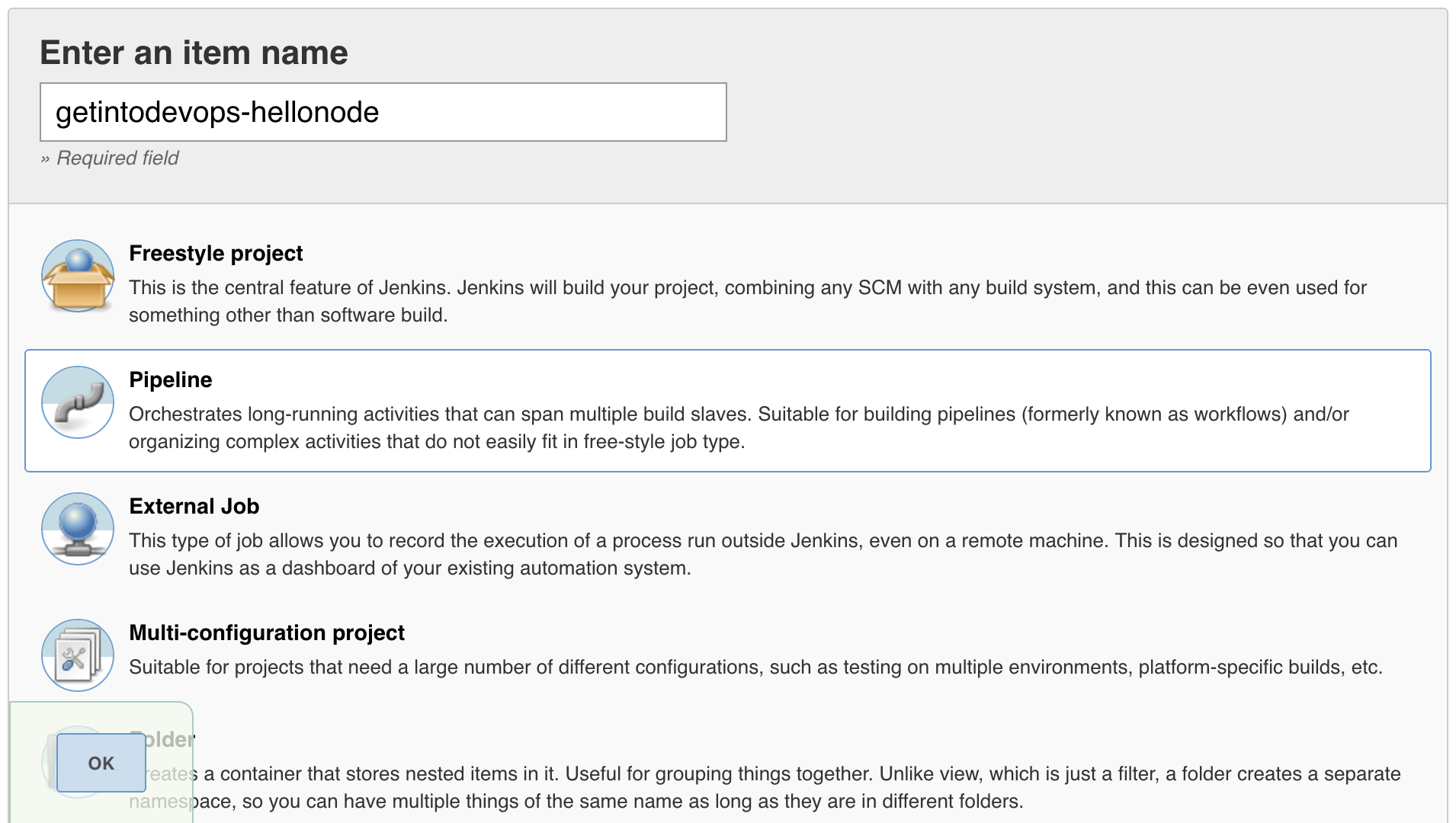
### **CREATING A JOB IN JENKINS**

The final thing we need to tell Jenkins is how to find our repository. We'll create a Pipeline job, and point Jenkins to use a Jenkinsfile in our repository.

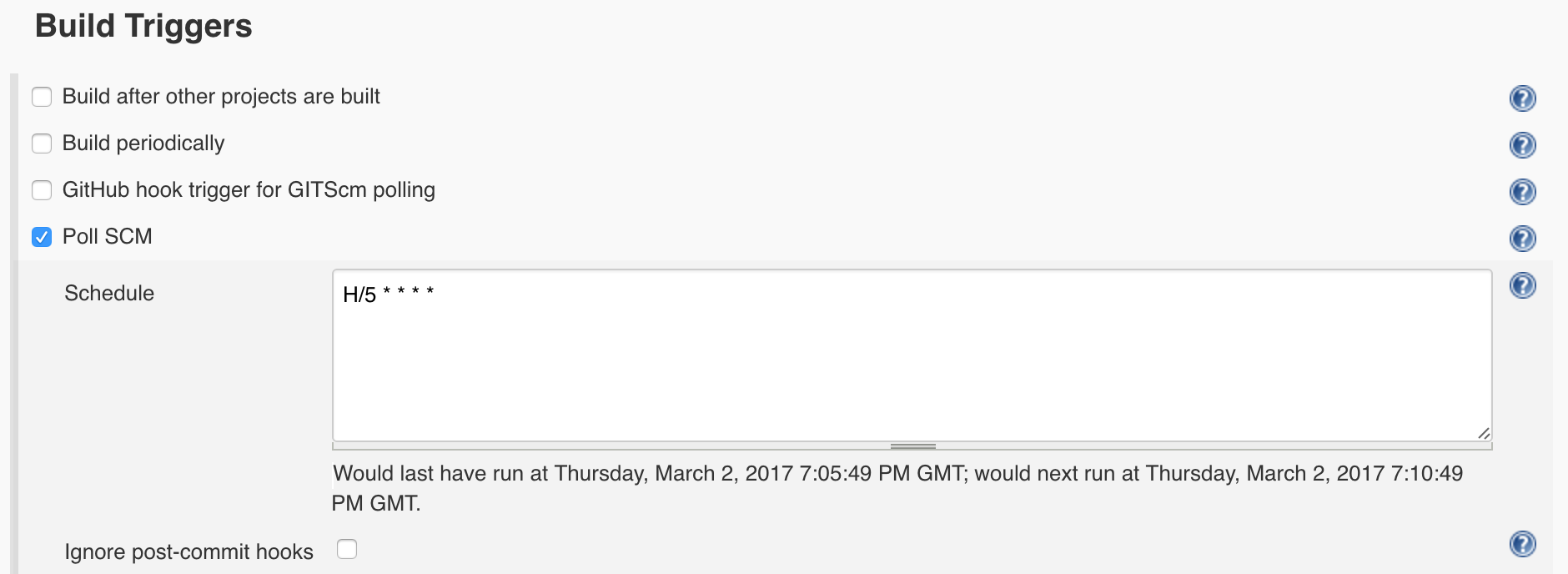
Here are the steps:



Click on **New Item** on the Jenkins front page.

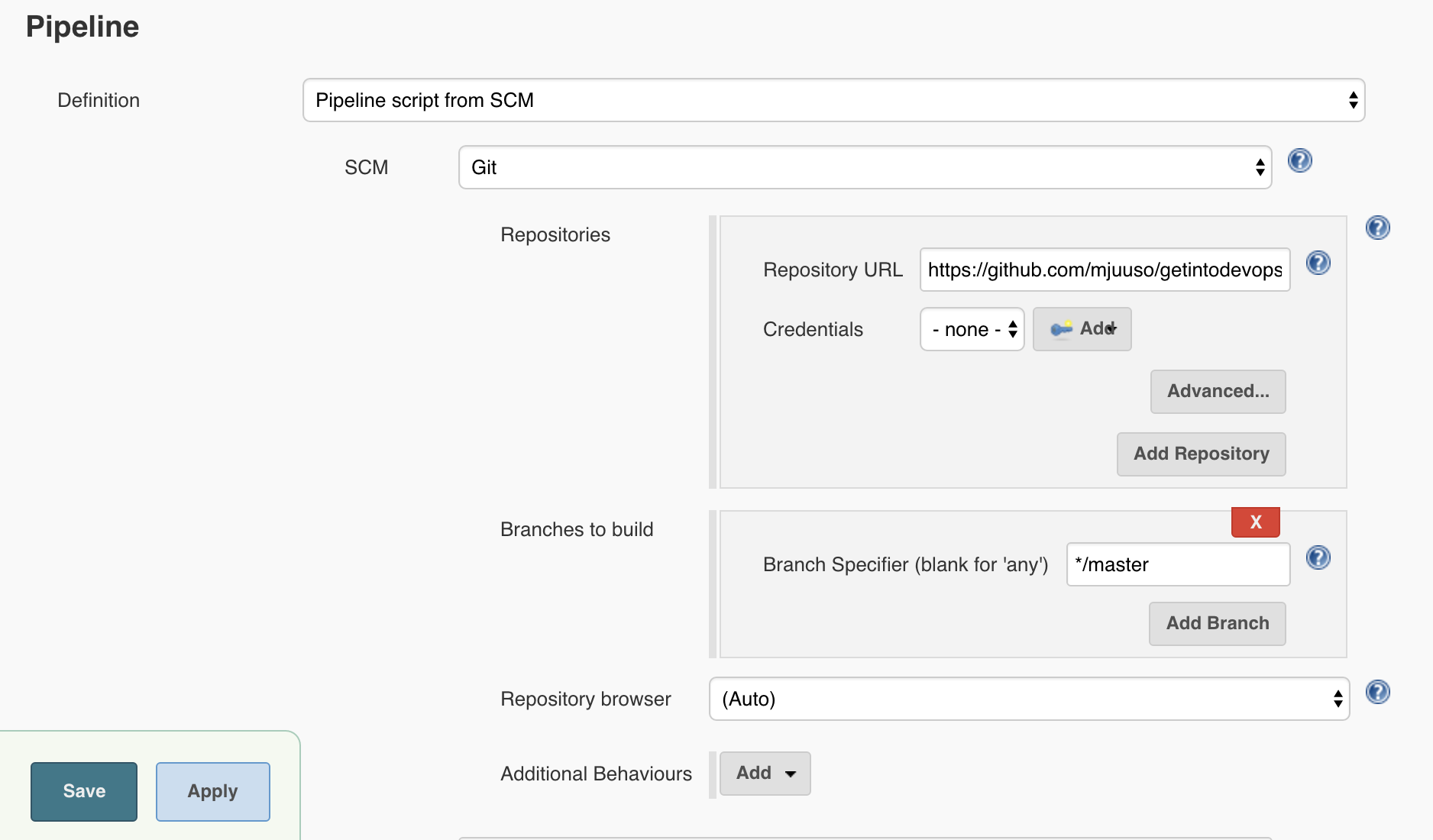


Type a name for your project, and select **Pipeline** as the project type.



Select **Poll SCM** and enter a polling schedule. The example here, H/5 \* \* \* \* will poll the Git repository every five minutes.

Note that I am polling for changes in this example because I am being lazy. If your repo is in Github, a much better approach is to [set up webhooks](https://jenkins.io/solutions/github/).



For the pipeline definition, choose **Pipeline script from SCM**, and tell Jenkins how to find your repository.

Finally, press **Save** and your pipeline is ready!

To build it, press **Build Now**. After a few minutes you should see an image appear in your Docker Hub repository, and something like this on the page of your new Jenkins job:

