**The Requirements**

We won’t be doing anything too language specific, so the following is the bare minimum of what you’ll need.

* Docker for Mac, Windows, or Linux
* Jenkins

For this example Jenkins and Docker will exist on the same machine. Specifically, they will exist and be able to interact on the host. This means Jenkins will not be a container. In a production scenario you might have Jenkins on a separate machine and the Docker host on a properly configured different machine. It doesn’t really matter, the concepts will still hold.

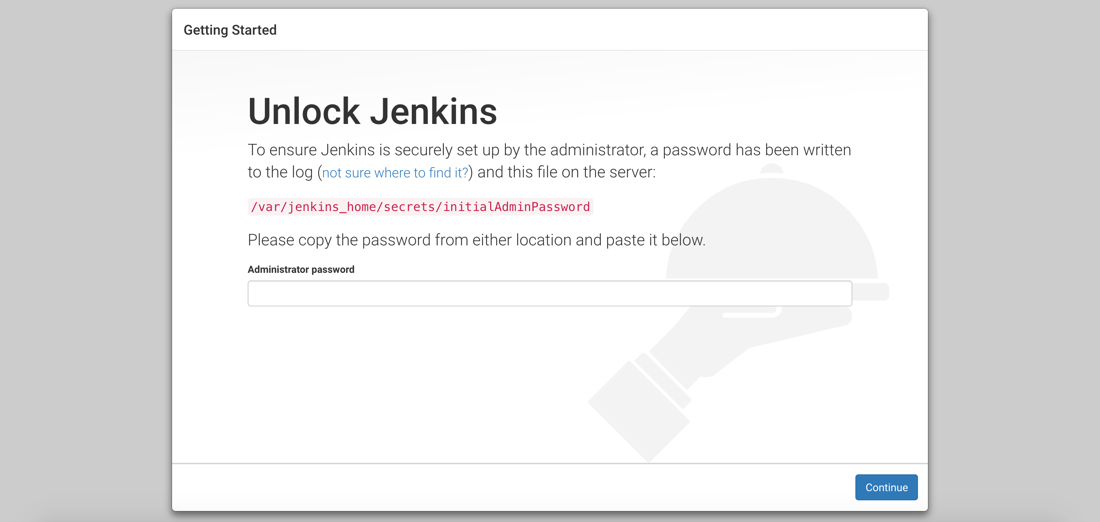
**Configuring Jenkins and Installing the Plugin Dependencies**

If you have Jenkins downloaded, but aren’t sure how to use it, don’t worry, we’ll walk through it. Chances are you have a WAR or JAR file right now. Launch Jenkins by executing the following:

java -jar jenkins.war -httpPort=8080

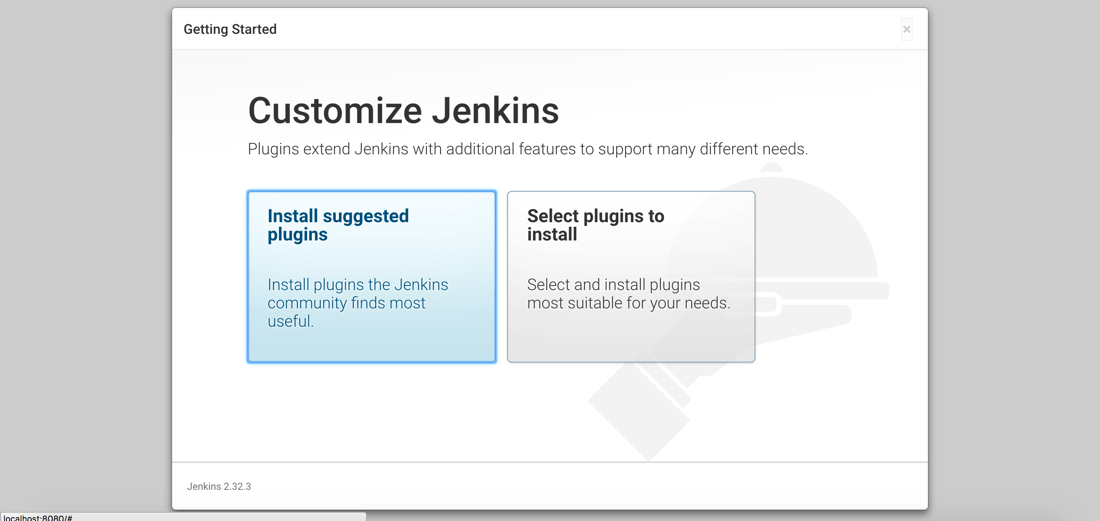
After a few minutes, Jenkins will be running. Make sure you note the generated password that shows up in the Terminal or Command Prompt as you’ll need it during the setup.

Navigate to **http://localhost:8080** in your browser to start the configuration process. The first thing you see should look something like the following.

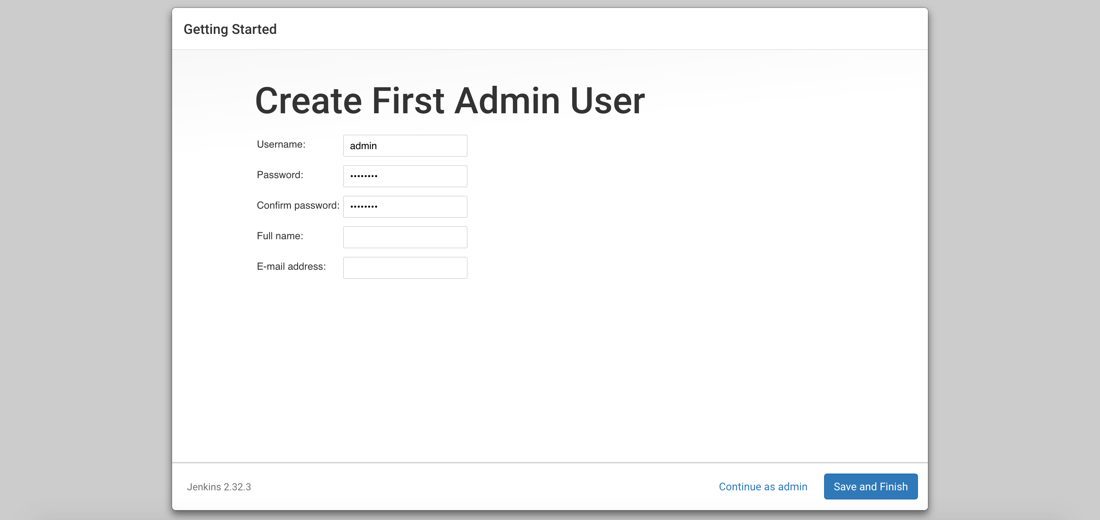


Go ahead and enter the generated password to move onto the next screen. The next screen should prompt you to decide how you want to install available plugins.

I recommend choosing to install the suggested plugins to make your life easier.



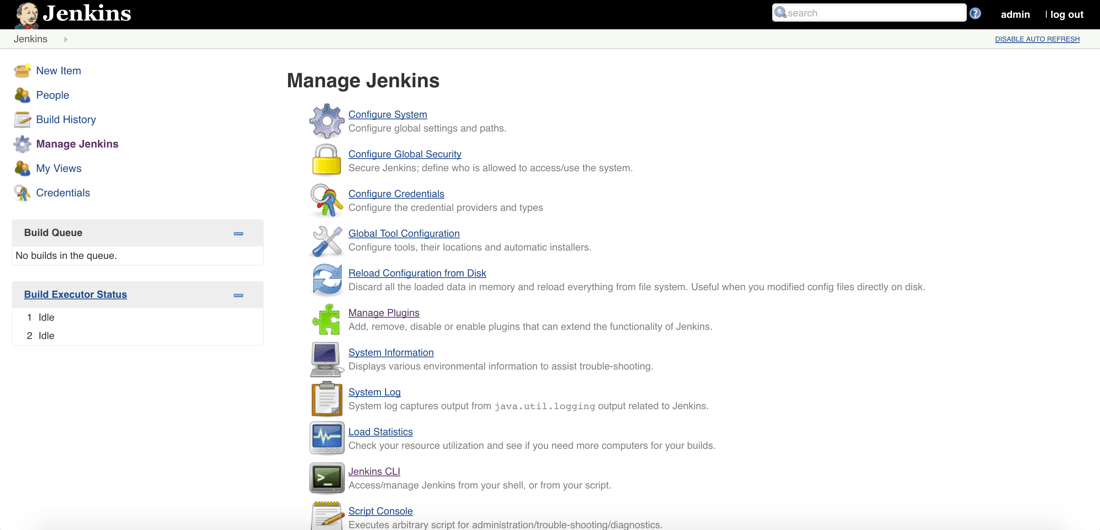
It may take some time to install all the plugins depending on your internet connection. Once the plugins have installed, you’ll be prompted to create an admin user.



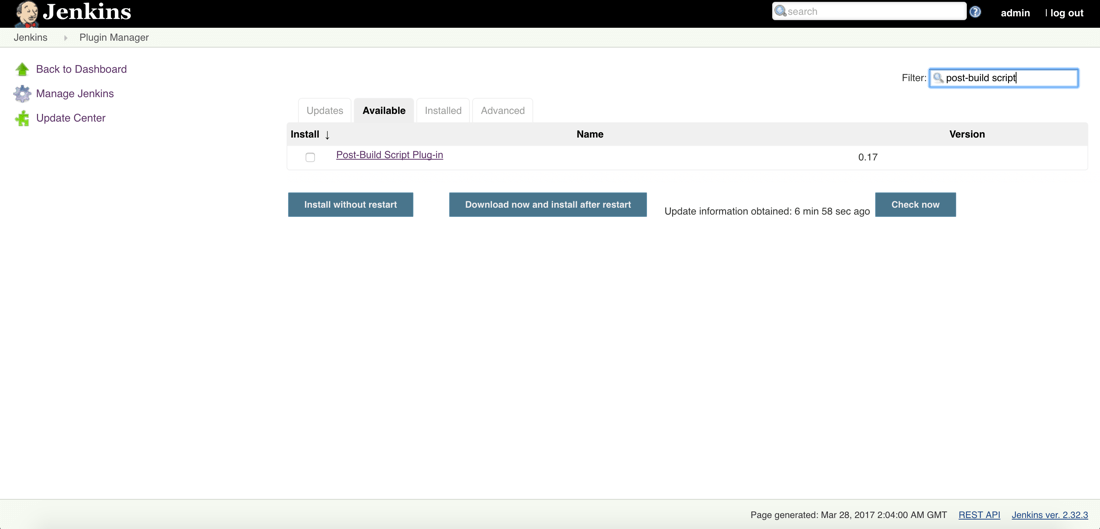
You can choose to create an administrative user or skip and use the generated password for everything.

At this point Jenkins is configured and ready to be used. You’ll notice that no jobs have been created and we won’t create any yet. First we want to install a missing plugin which will be used for the deployment of our containers.

Choose **Manage Jenkins** and then choose **Manage Plugins** from the list.



Search for **Post-Build Script Plug-in** from the list of available plugins. The reason we want this plugin is because it will allow us to run shell commands and scripts after the build step. In our example Jenkins exists on the host so it will have control over Docker. This means that we can deploy our container in the post-build step.

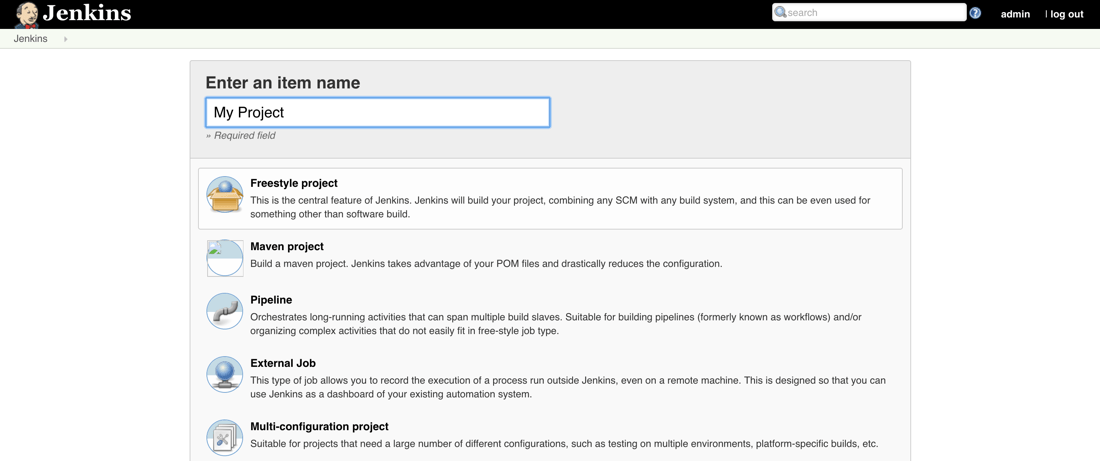


With the plugin installed, we can move onto creating our job. Remember, the job will download the code from Git, build the project, build a Docker image from the project, and deploy the image.

**Creating a Jenkins Job for Building and Deploying an Application via Docker**

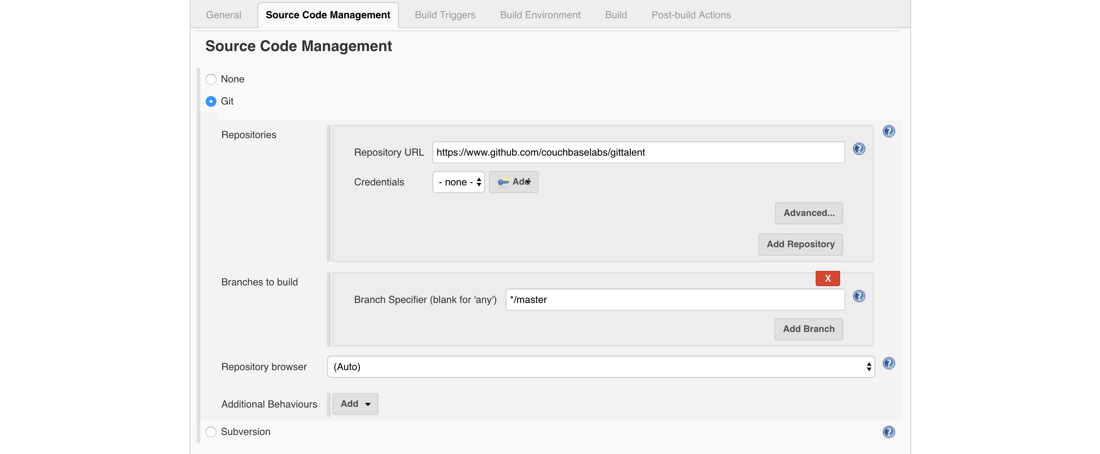
The point of this example isn’t in regards to developing an application. Any random GitHub project should work fine, but for this example I’m going to be referencing an Angular application I built a while back and uploaded.

From the root Jenkins screen, choose to create a new job.



We’ll be creating a **Freestyle Project** which will allow us to do anything. Give it a name and proceed to the next screen where we can configure our workspace source, build process, and deployment process.

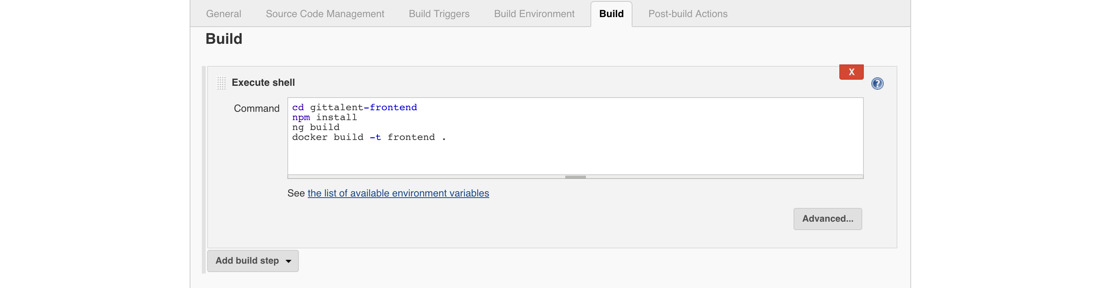
On the next screen find the **Source Code Management** section.



Here we’ll define the Git repository that we want to use. Like I said, I’m using a random project that I had once created. The catch here is that my project has a **Dockerfile** file in it to define how the Docker image is created. In case you’re curious, the repository I’m using is public and found at:

https://www.github.com/couchbaselabs/gittalent

After setting up the repository that will make up our workspace, we need to define the build process. Find the **Build** section of the screen.



A lot of magic happens in this part of the process. When the job executes, it pulls from the Git repository. My particular repository has a directory called **gittalent-frontend** which is what we’re going to build. It is an Angular project so we have to install all the dependencies first. Angular requires the Node Package Manager (NPM), so if you’re doing the same or similar, you’ll need Node.js available on the same host as Jenkins.

After the project is configured via NPM, we can use the development dependencies that NPM obtained to build the project with the Angular CLI. Now remember, the **gittalent-frontend** directory has a **Dockerfile** file in it. While not important for this example, it takes the built Angular project and copies it into the image when we build it.

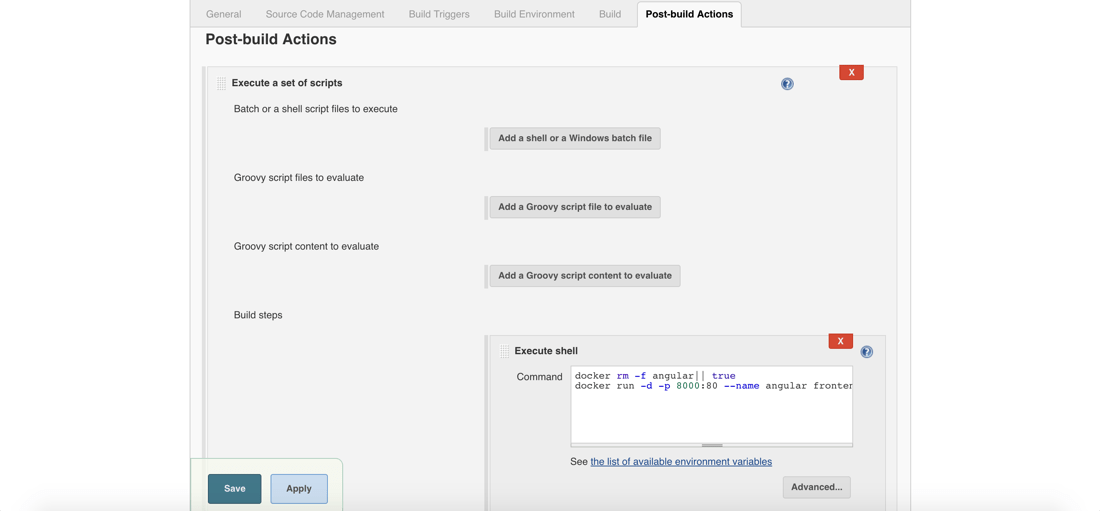
cd gittalent-frontend

npm install

ng build

docker build -t frontend .

Above are all the shell commands I’m using in my **Build** area, the final being the building of an image called **frontend**. Docker is running on the same host as Jenkins, so the image will be available.



The final part of our job will be to deploy what we’ve built. Remember we installed a plugin that lets us use shell commands and scripts. There are other plugins that you can download for SSH and other things, but it isn’t too relevant for us.

After the build process completes we want to execute a shell script as a post-build action.

docker rm -f angular || true

docker run -d -p 8000:80 --name angular frontend

Above is the post-build set of shell scripts that I’m using. We can’t just deploy the image that we’ve created because there may already be a container running. Instead we try to remove the container. Since the container may not exist, we are using pipes to force true and prevent failure regardless of the scenario.

After the container is removed, we can deploy our image as a new container.

We didn’t configure a GitHub hook to start our job when a commit happens, but you can if you want. Or you can trigger your job another way. Instead, just go ahead and choose **Build Now** from the job menu and watch it work.

**Conclusion**

You just saw how to use Jenkins and Docker to pull a project from GitHub or any other Git repository, build the project and a Docker image, and deploy that image as a container once complete. This was all done in an automated fashion called continuous deployment.

Now there are some changes you could make. For example, what if you wanted Jenkins to be a Docker container? How would you continuously deploy on the host in this scenario? While you could have a container control the Docker host, you probably shouldn’t, but if you really wanted to, you could try this:

docker run -d \

-v /var/run/docker.sock:/var/run/docker.sock \

-v $(which docker):/usr/bin/docker -p 8080:8080 myjenk