Building a Continuous Integration Pipeline with Docker

August 2015

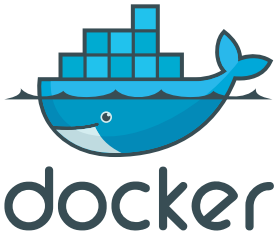


Table of Contents

Overview

Building a Continuous Integration Pipeline with Docker

Docker is the open platform to build, ship and run distributed applications, anywhere. At the core of the Docker solution is a registry service to manage images and the Docker Engine to build, ship and and run application containers. Continuous Integration (CI) and Continuous Delivery (CD) practices have emerged as the standard for modern software testing and delivery. The Docker solution accelerates and optimizes CI/CD pipelines, while maintaining a clear separation of concerns between the development and operations teams.

Integrating Docker into their CI pipeline has helped many organizations accelerate system provisioning, reduce job time, increase the volume of jobs run, enable flexibility in language stacks and improve overall infrastructure utilization. The BBC News was able to reduce the average job time by over 60% and increase their overall job throughput. ING Netherlands went from 9 months to push changes to pushing over 1,500 a week and using Docker to streamline their CD environment.

The immutability of Docker images ensures a repeatable deployment with what's developed, tested through CI, and run in production. Docker Engine deployed across the developer laptops and test infrastructure allows the containers to be portable across environments. Docker Trusted Registry (DTR) allows devops and release teams to manage those container images in a single location through the entire release process.

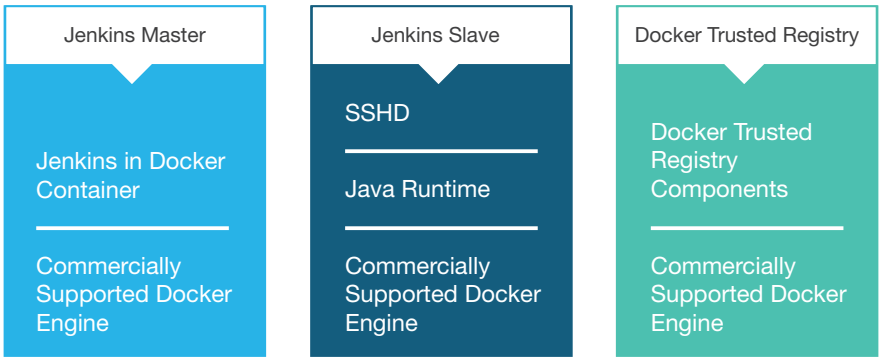
In this paper, we will discuss the construction of a CI pipeline consisting of Docker Engine, GitHub, Jenkins, and Docker Trusted Registry (DTR). The pipeline will be kicked off by a commit to a GitHub repository. The commit will cause Jenkins to run a build job inside a Docker container, and, upon successful completion of that job, push a Docker image up to Docker Trusted Registry.

If you are new to Docker, we suggest:

- Getting an introduction to Docker on our learning website training.docker.com
- Reading an overview of the Docker Solution docker.com/subscription and specifically the Docker Trusted Registry at docker.com/docker-trusted-registry
- Reviewing the product documentation docs.docker.com/docker-trusted-registry

Architectural Overview and Required Components

Architectural Components



Requisite compute resources:

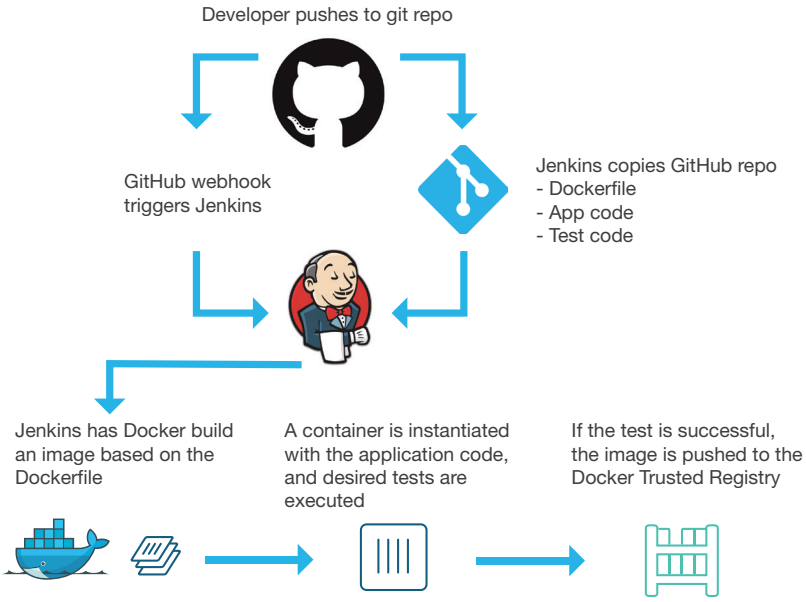
- Jenkins master running the commercially supported Docker Engine and the Jenkins platform in a Docker container
- Jenkins slave running the commercially supported Docker Engine with SSHD enabled and a Java runtime installed
- Docker host running the commercially supported Docker Engine and the Docker Trusted Registry components

Note: In a production environment there would likely be multiple slave nodes to handle workload scaling.

Additionally, there is a GitHub repository which hosts the application to be tested along with a Dockerfile to build the application node, and any code necessary for testing.

Finally, the Jenkins master takes advantage of the *GitHub Plugin*, which allows a Jenkins job to be triggered when a change is pushed to a GitHub repository

Workflow



The CI workflow described in this document is as follows:

1. Developer pushes a commit to GitHub
2. GitHub uses a webhook to notify Jenkins of the update
3. Jenkins pulls the GitHub repository, including the Dockerfile describing the image, as well as the application and test code.
4. Jenkins builds a Docker image on the Jenkins slave node
5. Jenkins instantiates the Docker container on the slave node, and executes the appropriate tests
6. If the tests are successful the image is then pushed up to Docker Trusted registry

Environment Prerequisites

Base installation and configuration of the various components are not covered in this reference architecture. It is assumed before beginning the steps outlined in this document the following prerequisites have been met:

Note: All nodes need to have the commercially supported Docker engine installed.

- Jenkins master running inside a Docker container on a dedicated Docker host. https://hub.docker.com/_/jenkins/
  - The GitHub plugin is also installed on the Jenkins master https://wiki.jenkins-ci.org/display/JENKINS/GitHub+Plugin
- Jenkins slave running on a dedicated host. https://wiki.jenkins-ci.org/display/JENKINS/Step+by+step+guide+to+set+up+master+and+slave+machines
- Docker Trusted Registry running on a dedicated host https://docs.docker.com/docker-trusted-registry/
  - DTR has the following prerequisites
    - Server (physical, virtual, or cloud) running a supported operating system (see https://www.docker.com/compatibility-maintenance)
    - Port 80 and 443 available to access the server
    - Commercially supported Docker Engine 1.6.x or higher installed (available as part of the Docker Subscription)
    - Adequate storage available for container images (recommended 5GB or more for initial deployment) based on one of the approved storage drivers (AWS S3 or Microsoft Azure are two such drivers)
    - An SSL certificate
    - Assigned hostname

- Docker Hub account (where license keys and downloads are available)
- Docker Subscription License Key

Note: To obtain a Docker Subscription trial license visit https://hub.docker.com/enterprise/trial/

- All components are able to communicate to each other on the network, including domain name resolution and all the necessary certificates have been installed.

See below for a list of the network ports and protocols that need to be accessible between hosts.

Open Network ports and communication protocols

Purpose	Source	Destination (service:port)	configurable destination port	Application Protocol
Image Push/Pull	Docker Engine	DTR_Server: 443	Yes	HTTPS
Pulling Audit Logs	3rd party Logging System	DTR_Server: 443	Yes	HTTPS
Admin UI	Client Browser	DTR_Server: 443	No	HTTPS
Identity Provider Service	DTR (Auth Server)	Directory_Server: 389 Directory_Server: 636	No	LDAP or Start TLS w/ LDAP LDAP(S)

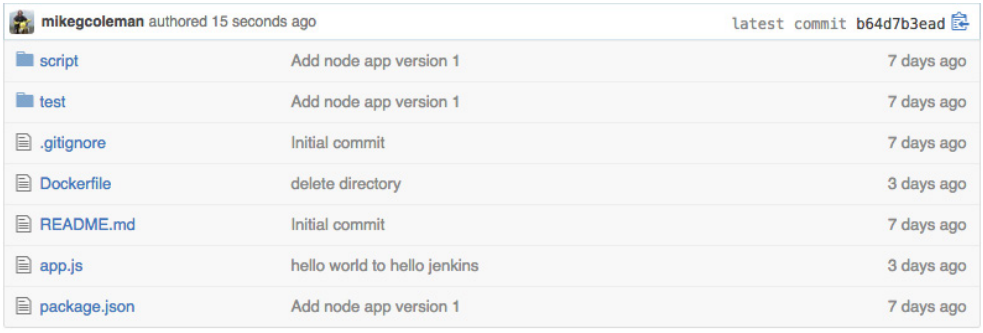
**configurable destination port* means the destination port can be changed to any port number while using the same application protocol.

Configuring GitHub

In order to create and test a target application, the GitHub repository for the target application needs to contain the following components:

- The application code
- The test code for the application
- A Dockerfile that describes how to build the application container, and copies over the necessary application and test code.

As an example, the following image depicts the repository for a very simple Node.js application:



The Dockerfile is constructed as follows:

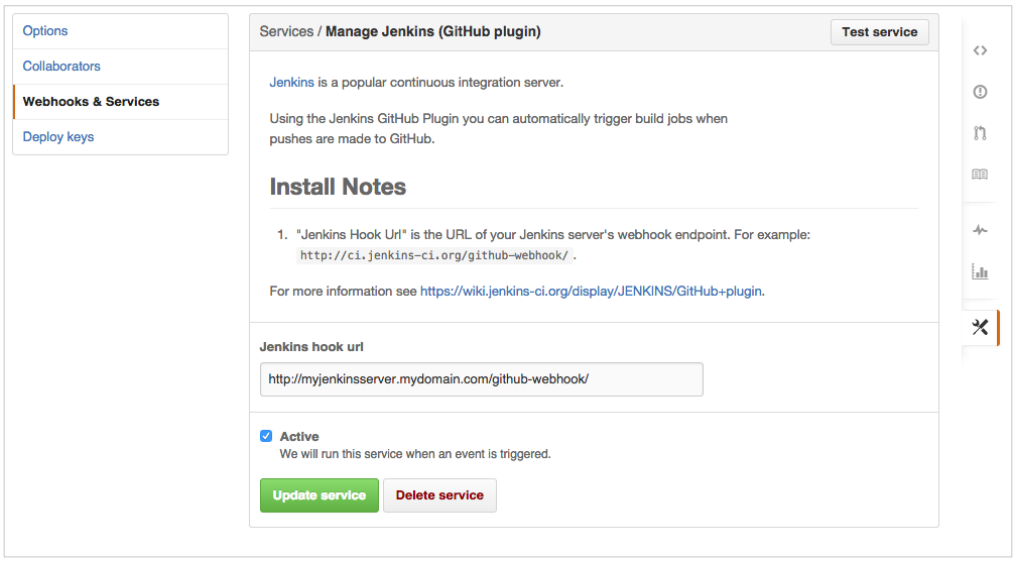
```
FROM node:latest
MAINTAINER mike.coleman@docker.com

# set default workdir
WORKDIR /usr/src

# Add package.json to allow for caching
COPY package.json /usr/src/package.json

# Install app dependencies
RUN npm install

# Bundle app source and tests
COPY app.js /usr/src/
COPY test /usr/src/test
COPY script /usr/src/script

# user to non-privileged user
USER nobody

# Expose the application port and run application
EXPOSE 5000
CMD ["node","app.js"]
```

This Dockerfile will pull the supported Node.js image from the Docker Hub. Next, it will install any necessary dependencies (based on the package.json file), and then copy the application code and test files from the GitHub repository to the /src directory in the image. Finally, it will expose the appropriate network port, and execute the application.

Note: *This Dockerfile is presented as an example, there is nothing in the file specific to enabling the CI workflow. The only requirement for this workflow is that there is a Dockerfile that will build the requisite image needed for testing.*

Note: *The tests for the application are included in the /script and /test directories.*

In addition to the presence of a Dockerfile, the repository needs to be configured to notify the Jenkins server when a new commit happens. This is done via the "Webhooks and Services" section of the "Settings" page.



Configuring the Jenkins Master

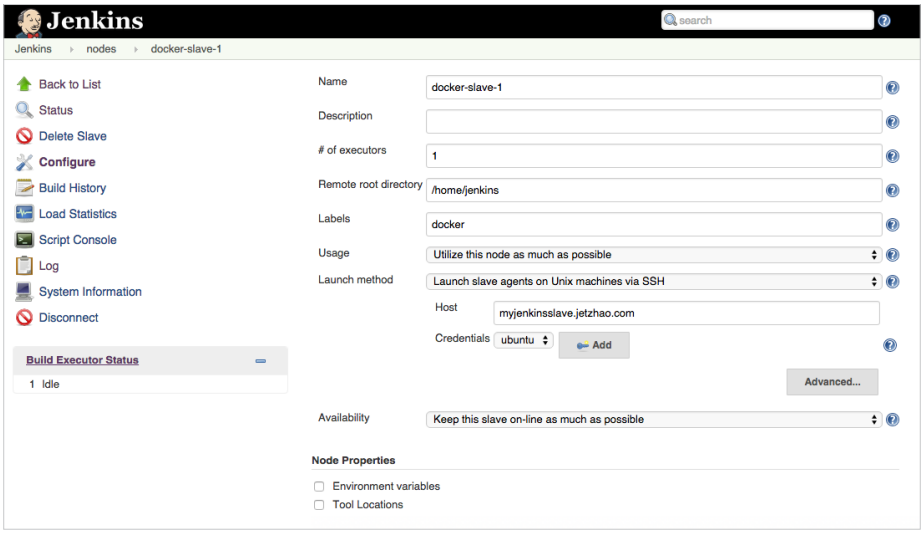
After the base Jenkins image is installed, and the service is up and running the GitHub Plugin needs to be installed on the Jenkins master. This plugin allows for a Jenkins job to be initiated when a change is pushed a designated GitHub repository

Configuring the Jenkins Slave

In addition to the Jenkins master, there needs to be at least one Jenkins slave configured. Since the slave will run all of its test builds inside of Docker containers, there are no requirements around installing specific languages or libraries for any of the applications that will be tested.

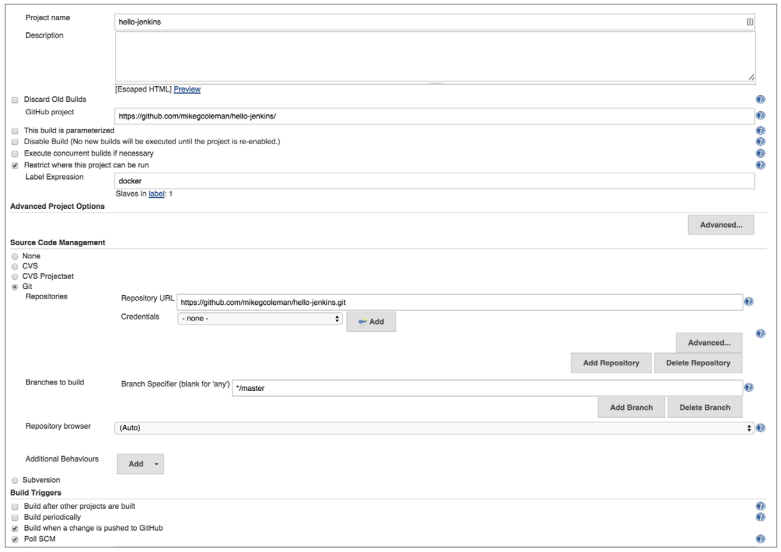
However, the slave does need to run the commercially supported Docker Engine (https://docs.docker.com/docker-trusted-registry/install/), along with having SSH enabled, and a Java runtime installed (the last two requirements are standard with Linux Jenkins slaves).

It's also helpful to tag the slave with a name so that it can easily be identified as a Docker host. In the screen shot below, the slave had been tagged "docker".



Configuring the test job

Once the slave has been added, a Jenkins job can be created. The screen shot below shows a Jenkins job to test the sample Node.js application.



The key fields to note are as follows:

- "Restrict where this build can run" is checked, and the label "docker" is supplied. This ensures the job will only attempt to execute on slaves that have been appropriately configured and tagged.
- Under "Source Code Management" the name of the target GitHub repository is supplied. This is the same repository that houses the Dockerfile to build the test image, and that has been configured to use the GitHub webhook. Also ensure that "Poll SCM" is checked (it is not necessary to set a schedule).
- Under "Build Triggers" "Build when a change is pushed to GitHub" instructs Jenkins to fire off a new job every time the webhook sends a notification of a new push to the repository.

Under the "Build" section of the Jenkins job we execute a series of shell commands:

```
# build docker image
docker build --pull=true -t dtr.mikegcoleman.com/hello-jenkins:$GIT_COMMIT .
# test docker image
docker run -i --rm dtr.mikegcoleman.com/hello-jenkins:$GIT_COMMIT ./script/test
# push docker image
docker push dtr.mikegcoleman.com/hello-jenkins:$GIT_COMMIT
```

The first command builds a Docker image based on the Dockerfile in the GitHub repository. The image is tagged with the git commit id (the environment variable is provided courtesy of the GitHub plugin).

The command instantiates a new container based on the image, and executes the test scripts that were copied over during the image build.

Finally, the image is pushed up to our Docker Trusted Registry instance.

Note: *Each step only executes if the previous step was successful. If the test run does not succeed, then the image would not be copied to the Docker Trusted Registry instance.*

One important consideration is how the storage of build images will be handled on the Jenkins slave machines. There is a tradeoff that administrators need to weigh.

On one hand, leaving the build images on the server increases the speed at which subsequent jobs execute because Jenkins will not need to pull the entire image down from the registry.

However, even though Docker minimizes storage impact by using union file systems to share layers between images, if left unchecked the library of build images could consume large amounts of space. The amount of space is entirely dependent on how many different images are being used, and the size of those images.

In the end, the tradeoff comes down to build speed vs. storage consumption. Organizations will need to determine what the right mix is for them.

Putting it all together: Running a Test

To kick off the workflow a developer makes a commit to the applications GitHub repository. This fires off a GitHub webhook, which notifies Jenkins to execute the appropriate tests.

Jenkins receives the webhook, and builds a Docker image based on the Dockerfile contained in the GitHub repo on our Jenkins slave machine. After the image is built, a container is created and the specified tests are executed.

If the tests are successful, the validated Docker image is pushed to the Docker Trusted Registry instance.

Conclusion

Organizations who leverage Docker as part of their continuous integration pipeline can expect to increase stability, reduce complexity, and increase the agility of their software development processes. Docker allows users to ensure consistent repeatable processes, while simultaneously allowing for a reduction in the number of test images that need to be tracked and maintained. The lightweight nature of containers means they can be spun up faster, allowing for more rapid test cycles.

Additional Resources

- Docker Trusted Registry www.docker.com/docker-trusted-registry
- Documentation docs.docker.com/docker-trusted-registry
- Case Studies www.docker.com/customers

www.docker.com