Docker Custom Build Environment Plugin

Introduction

A major requirements for a Jenkins-based Continuous Integration/Delivery setup are:

- Ensure the build/test environment is exactly reproducible from a pre-defined setup.
- Ensure the build/test environment is well isolated from other builds executed on the same infrastructure.

Docker is a great way to bootstrap reproducible and isolated environments. Compared to a virtual machine, it's faster to launch and lighter to use. Another benefit is the Docker image can be defined both as a binary image pulled from repository, or as a plain text Dockerfile you can store in SCM along side the project source code, so that the source and environment are always in sync and recorded.

CloudBees Docker Custom Build Environment Plugin has been designed to make Docker Images / Dockerfile a first class citizen in a continuous delivery setup, which allows the simplest way for a development team to manage the exact bits of the build environment whilst the infrastructure team only has to focus on available resources hosting arbitrary docker containers.

User Guide

CloudBees Docker Custom Build Environment Plugin can be used from any job type, it appears in the "build environment" section and let you configure your build to run inside a Docker container.

You can:

- Select the Docker image to run the build as a Docker image to be pulled. This is comparable to the docker-plugin approach to offer docker slaves within Jenkins, but without any prerequisites on the Docker image nor need for Administrator privileges to configure the adequate slave template.
- Configure the plugin to build a container image from a Dockerfile stored in project repository. With this setup, you get both the project source code and build environment defined in SCM. This is my preferred way to use this plugin.

SCM checkout will run within a classic Jenkins slave execution context - this is required to access the project Dockerfile then build and run the required container.

Using Docker image

CloudBees Docker Custom Build Environment let you use arbitrary docker image to host the build. You can use such an image you build on your own (or using CloudBees Docker Build and Publish plugin) to define the set of prerequisites for your project and share with the development team, as well as reuse for your CI job. In following sample, maven+jdk8 image available on DockerHub is used as a reference environment to host a maven build.
Using a Dockerfile

You also can have a Dockerfile and context stored in your project SCM aside your source code, so team can maintain them together and/or use distinct Dockerfile per branch. If you use this option, CloudBees Docker Custom Build Environment will build the image on first build and then will compare your Dockerfile with the one used on previous build to check a rebuild is necessary.

As for Docker images, there's no requirement on the Dockerfile you're using.

Advanced options

CloudBees Docker Custom Build Environment integrates with Docker-commons plugin so you can define the docker cli executable to be used to interact with Docker daemon, as well as select TLS credentials to be used to access a secured infrastructure (which is highly recommended).

- **Verbose** option allows to dump on build log the docker-cli command output. This is mostly designed for diagnostic purpose.

Volumes

The build container is ran with entire Jenkins Home mounted inside container, so your build tools can access the SCM working copy and produce binaries / reports. The temporary directory is also mounted as many plugin do rely on this one to pass commands or credentials files that get deleted on build completion.

You can configure extra directories to be mounted into container, a common use-case is to have the dependency manager local cache storing artifacts on executor host so you don't have to download them again and again for every build.

User Group

Plugin design do require the user running build commands to be the same as Jenkins slave agent user, so Jenkins can manage files created in Workspace without hitting permission issues. As Docker does not (yet) support user namespace, there's no way to use a distinct user without breaking workspace at some point. For some use cases this is a major issue, as the container is designed with some specific permission set to a user.

As a workaround, you can define the group for the user running commands during the build, and setup this group inside your container with adequate permissions.

Docker in Docker
Sometime your build will require to create some other Docker containers, for sample to bootstrap a test database used by your integration tests.

Docker allows to host a docker daemon inside a docker container, known as "Docker in Docker". You can use jpetazzo/dind docker image (or a derived one) but will need to run container with escalated permission ("privileged mode") which you should consider twice as this has some significant security impacts. This also require some tweak in plugin configuration, so the command used to bootstrap the container do setup the docker daemon:

- run in privileged mode
- set startup command to run wrapdocker
- configure plugin to run within the 'docker' group so you can access the docker socket.

An alternative to Docker in Docker is to let the docker container hosting the build communicate with the host docker daemon so it can start other container, aside the build container. To achieve this, you can configure your Build container to run with docker daemon unix socket mounted, so you can run docker commands within your build. Use the "volumes" advanced option for this purpose.

Please note : this is your responsibility to get the docker cli executable in your container. Simplest option is to include those lines in your Dockerfile:

```yaml
RUN wget https://get.docker.io/builds/Linux/x86_64/docker-latest -O /bin/docker
RUN chmod +x /bin/docker
```

Within a build step in your job you can then run docker containers as you would from command line. If the container you're starting need to share files with the build, you can configure it to mount the same volumes used by your build container. Let's say for sample you have a docker image for application server which expect as launch parameter the path to your application WAR package. You can then run:

```bash
docker run --volumes-from $BUILD_CONTAINER_ID application_server_docker_image $WORKSPACE/dist/myapp.war
```

**Environment variables**

plugin do expose BUILD_CONTAINER_ID variable during the build so you can make reference to the container hosting the build when needed.

**Open JIRA issues**

<table>
<thead>
<tr>
<th>T</th>
<th>Key</th>
<th>Summary</th>
<th>Assignee</th>
<th>Reporter</th>
<th>P</th>
<th>Status</th>
<th>Resolution</th>
<th>Created</th>
<th>Updated</th>
<th>Due</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JENKIN S-35025</td>
<td>Container command <code>/bin/cat</code> not found or does not exist</td>
<td>Unassigned</td>
<td>Cedric Thiebault</td>
<td>![Status Icon]</td>
<td>OPEN</td>
<td>Unresolved</td>
<td>May 23, 2016</td>
<td>Apr 24, 2019</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JENKIN S-40684</td>
<td>Can't add a shell build step with docker</td>
<td>Unassigned</td>
<td>Nicolas Poirey</td>
<td>![Status Icon]</td>
<td>OPEN</td>
<td>Unresolved</td>
<td>Dec 27, 2016</td>
<td>Aug 05, 2019</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JENKIN S-29621</td>
<td>EnvInject Plugin is only working to a limited extent in combination with the docker build environment</td>
<td>Unassigned</td>
<td>Michael Süß</td>
<td>![Status Icon]</td>
<td>OPEN</td>
<td>Unresolved</td>
<td>Jul 24, 2015</td>
<td>Apr 24, 2019</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JENKIN S-29677</td>
<td>Other build environment plugins aren't compatible with the Docker Build Environment Plugin</td>
<td>Jon Hermansen</td>
<td>Michael Süß</td>
<td>![Status Icon]</td>
<td>OPEN</td>
<td>Unresolved</td>
<td>Jul 28, 2015</td>
<td>Feb 18, 2018</td>
<td></td>
</tr>
</tbody>
</table>
Prerequisites

The slave executor(s) running jobs need to have docker installed and the daemon running. We suggest you use a "docker" label for such slaves, so you benefit from Jenkins slave management and cloud capabilities.

History
This plugin was already known as "Oki-Docki" but this name made it difficult for people to discover it within update center :-\.

It has been created as part of DockerCon hackathon by Nicolas and Yoann. We ended as #2 team on the challenge :D

**Developer tips**

If you want to run this plugin on Windows / OSX for development, please note the plugin will bind mount the temporary directory inside container, so you probably will have to run jenkins JVP with `-Djava.io.tmpDir=$HOME/tmp` as only the users home directory is accessible when using boot2docker.

**Future plans**

- support docker-compose

**Versions**

**Not released yet**

1.6.4

- Fixed a NPE for jobs created prior to 1.6.2 ([JENKINS-31220](#))

1.6.3

- Fixed a regression introduced in 1.6.2, corrupting environment variables ([JENKINS-31166](#))

1.6.2

- Support bridge 'net' flag ([commit](#))
- Do not append command if not set ([JENKINS-30692](#)) (#33)
- Added an option to force pull the image
- Expose build wrappers contributed environment variables
- Ensure docker0 is up before trying to resolve it

1.6.1

- Use the Java API to lookup docker0 ip ([JENKINS-30512](#)) (#32)
- Add buildwrapper environment variables to the docker context

1.6

- support maven job type
- expose dockerhost IP as "dockerhost" in /etc/host
- configure container with a subset of build environment, as slave node environment doesn't make sense inside container.

1.5

- Option to configure volumes
- plugin now can run docker-in-docker and comparable advanced use-cases

1.4

- Support Node Properties environment variables (to define DOCKER_HOST per node for example)

1.3

- Support use of alternate Dockerfile
- Allows to run containers from the container hosting the build (see "Docker in Docker")
- Expose build container identifier as BUILD_CONTAINER_ID environment variable
- Code cleanup

1.2

- Initial release as "CloudBees Docker Custom Build Environment Plugin" (plugin was previously known as "Oki-Docki")

**Notes**

**Implementation details**

The docker container is ran after SCM has been checked-out into a slave workspace, then all later build commands are executed within the container thanks to `docker exec` introduced in Docker 1.3. When configured to build container from a Dockerfile, the plugin computes the Dockerfile checksum and uses it as container ID, so it can detect if the image exists on a slave and so only build it the first time it is requested.
Comparison

Compared to docker plugin,

- This plugin can use arbitrary docker images, there is NO prerequisite to get a specific user set, ssh daemon, or even JDK available in docker container you use for the build - no need for CI-specific docker image, can use the exact same docker image you use on developer workstation to run/test your project.
- Changes to the project that require new tools / version upgrades can be reflected in the Dockerfile within an atomic commit. No need to reconfigure the job or wait for the adequate slave to be setup. You can also use a distinct Dockerfile per project branch.
- The user doesn't need Administrator privileges to setup a docker-slave template, you just need to commit a Dockerfile to your source repository.
- Docker-plugin abuses the Jenkins Cloud API. i.e. you have to define a fixed IP address and can't benefit from a Cloud slave provider, or a pool of generic slaves. CloudBees Docker Custom Build Environment only relies on slaves which have docker installed, and Jenkins will provision/pick-up available ones using all available slaves provider plugins.