Introduction

The Docker Container as a Service (CaaS) platform delivers a secure, managed application environment for developers to build, ship, and run enterprise applications and custom business processes. In the “build” part of this process, there are design and organizational decisions that need to be made in order to create an effective enterprise development pipeline.

What You Will Learn

In an enterprise, there can be hundreds or even thousands of applications developed by in-house and outsourced teams. Applications can be deployed to multiple heterogeneous environments (development, test, UAT, staging, production, etc.), each of which can have very different requirements. Packaging an application in a container with its configuration and dependencies guarantees that the application will always work as designed in any environment. The purpose of this document is to provide you with typical development pipeline workflows as well as best practices for structuring your development process using Docker EE (formerly known as Docker Datacenter).

In this document you will learn about the general workflow and organization of the development pipeline and how Docker EE integrates with existing build systems. It also covers the specific developer, CI/CD, and operations workflows and environments.

General Organization

A typical enterprise has separate development and operations teams and may have some level of DevOps adoption. In general, operations teams are responsible for delivering and supporting the infrastructure up to the operating systems and even middleware components. Development teams are responsible for building and maintaining the applications. There is also some type of continuous integration (CI) for automated build and testing as well as continuous delivery (CD) for deploying versions to the different environments.

General Workflow

A typical CI/CD workflow is shown in the following diagram:
It starts on the left-hand side with development teams building applications. A CI/CD system then runs unit tests, packages the applications, and builds Docker images on the Docker Universal Control Plane (UCP). If all tests pass, the images can be signed using Docker Content Trust (https://docs.docker.com/datacenter/ucp/2.0/guides/content-trust/) and shipped to Docker Trusted Registry (DTR). The images can then be run in other non-production environments for further testing. If the images pass these testing environments, they can be signed again and then deployed by the operations team to the production environment.

**UCP Clusters**

Enterprises typically have separate production and non-production UCP clusters as previously shown. This is a natural fit with existing infrastructure organization and responsibilities. Enterprises typically have a production environment with higher security requirements, restrained operator access, a high-performance infrastructure, high-availability configurations, and full disaster recovery with multiple data centers. The non-production environment has different requirements with the main goal being testing and qualifying applications for production. The interface between the non-production and production clusters is DTR.

The question of whether to have a separate UCP cluster per availability zone or have one "stretched cluster" mainly depends on the network latency and bandwidth between availability zones. There could also be existing infrastructure and disaster recovery considerations to take into account.

In an enterprise environment where there can be hundreds of teams building and running applications, a best practice is to separate the build from the run resources. By doing this, the image building process does not affect the performance or availability of the running containers/services.

There are two common methods of building images using Docker EE:

- **Developers build images on their own machines then push them to DTR.** - This is suitable if there is no CI/CD system and no dedicated build cluster. Developers have the freedom to push different images to DTR.

- **A CI/CD process builds images on a build cluster and pushes them to DTR.** - This is suitable if an enterprise wants to control the quality of the images pushed to DTR. Developers commit Dockerfiles to version control. They can then be analyzed and controlled for adherence to corporate standards before the CI/CD
system builds the images, tests them, and pushes them to DTR. In this case, CI/CD agents should be run directly on the dedicated build nodes.

**Note:** In the CI/CD job, it is important to insure that images are built and pushed from the same Docker node so there is no ambiguity in the image that is pushed to DTR.

### DTR Clusters

Unlike the separate production and non-production UCP clusters, enterprises commonly have a single master DTR cluster. This allows enforcement of enterprise processes such as Security Scanning [here](https://docs.docker.com/docker-cloud/builds/image-scan/) in a centralized place. If pulling images from globally distributed locations takes too long then you can use the DTR Content Cache [here](https://docs.docker.com/datacenter/dtr/2.2/guides/admin/configure/deploy-a-cache/) feature to create local caches.

**Note:** Policy enforcement on image signing [here](https://docs.docker.com/datacenter/ucp/2.0/guides/content-trust/) will not currently work if you have your DTR in a separate cluster from UCP.

### Developer Workflow

Developers and application teams usually maintain different repositories within the organization to develop, deploy, and test their applications. This section discusses the following diagram of a typical developer workflow using Docker EE as well as their interactions with the repositories:

A typical developer workflow follows these steps:

1. **Develop Locally** - On the developer's machine or environment the developer locally builds images, run containers, and test their containers. There are several types of files and their respective repositories that are used to build Docker images.
   - **Version Control** - This is used mainly for text-based files such as Dockerfiles, `docker-compose.yml`, and configuration files. Small binaries can also kept in the same version control. Examples of version control...
control are git, svn, Team Foundation Server, VSTS, and Clear Case.

- **Repository Manager** - These hold larger binary files such as Maven/Java, npm, NuGet, and RubyGems. Examples include Nexus, Artifactory, and Archiva.

- **Package Repository** - These hold packaged applications specific to an operating system such as CentOS, Ubuntu, and Windows Server. Examples include yum, apt-get, and PackageManagement. After building an image, developers can run the container using the environment variables and configuration files for their environments. They can also run several containers together described in a `docker-compose.yml` file and test the application.

2. **Push Images** - Once an image has been tested locally, it can be pushed to the Docker Trusted Registry. The developer must have an account on DTR and can push to a registry on their user account for testing on UCP. For example: `docker push dtr.example.com/kathy.seaweed/apache2:1.0`

3. **Deploy on UCP** - The developer might want to do a test deployment on an integration environment on UCP in the case where the development machine does not have the ability or access to all the resources to run the entire application. They might also want to test whether the application scales properly if it's deployed as a service. In this case the developer would use CLI-based access (https://docs.docker.com/datacenter/ucp/2.0/guides/access-ucp/cli-based-access/) to deploy the application on UCP (https://docs.docker.com/datacenter/ucp/2.0/guides/applications/deploy-app-cli/).

   Use `$ eval $(<env.sh)` to point the Docker client to UCP. Then run the following commands: `$ docker run -dit --name apache2 dtr.example.com/kathy.seaweed/apache2:1.0` `$ docker-compose --project-name wordpress up -d` `$ docker service create --name apache2 dtr.example.com/kathy.seaweed/apache2:1.0`

4. **Test the Application** - The developer can then test the deployed application on UCP from his machine to validate the configuration of the test environment.

5. **Commit to Version Control** - Once the application is tested on UCP, the developer can commit the files used to create the application, its images, and its configuration to version control. This commit triggers the CI/CD workflow.

### Developer Environment and Tools

A developer's machine should have an edition of Docker (Windows (https://docs.docker.com/docker-for-windows/), Mac (https://docs.docker.com/docker-for-mac/), or Linux (https://docs.docker.com/engine/installation/linux/)) installed on it. The installation provides Docker engine (https://docs.docker.com/engine/userguide/intro/), Docker CLI client (https://docs.docker.com/engine/reference/commandline/cli/), Docker Compose (https://docs.docker.com/compose/overview/), and Docker Notary command line (https://docs.docker.com/notary/getting_started/).

In some enterprises, the version of the operating system used on developer machines is older and not compatible with Docker for Windows or Mac. In this case Docker Toolbox (https://docs.docker.com/toolbox/overview/) can be used.

### IDE

Docker versions do not provide a native IDE for developing with Docker. The primary interface is the command line API. However, most leading IDEs (NetBeans, Eclipse, IntelliJ, Visual Studio) have some support for Docker through plugins or add-ons. Our labs (https://github.com/docker/labs/tree/master/developer-tools) contain tutorials on how to setup and use common developer tools and programming languages with Docker.

---

**Tip:** Optimizing images sizes. If an image size becomes too large, a quick way to identify where possible optimizations are is to use the `docker history <image>` command. It will tell you which lines in the Dockerfile added what size to the image.

### Docker Client CLI Contexts
When working with Docker EE and the Docker command line, it is important to keep in mind the context that the command is running in.

- **Local Docker Engine** — This is the main context used for development of Dockerfiles and testing locally on the developer's machine.

- **Remote UCP CLI** — CLI-based access ([https://docs.docker.com/datacenter/ucp/2.0/guides/access-ucp/cli-based-access/](https://docs.docker.com/datacenter/ucp/2.0/guides/access-ucp/cli-based-access/)) is used for building and running applications on a UCP cluster.

- **UCP GUI** — The Docker EE web user interface provides an alternative to the CLI.

- **Remote UCP Node** — Sometimes it can be useful for debugging to directly connect to a node on the UCP cluster. In this case, SSH can be used, and the commands are executed on the Docker engine of the node.

A quick way to see which context you are in is by doing an `export` command. If you have a `DOCKER_HOST` environment variable set, then you have most likely configured your UNIX session to use the UCP client bundle, meaning all Docker commands are executed on the UCP cluster.

## CI/CD Workflow

A CI/CD platform uses different systems within the organization to automatically build, deploy, and test applications. This section discusses a typical CI/CD workflow using Docker EE and the interactions with those repositories as showing in the following illustration:

A typical CI/CD workflow follows these steps:

1. **Build Application** — A change to the version control of the application triggers a build by the CI Agent. A build container is started and passed parameters specific to the application. The container can run on a separate Docker host or on UCP. The container obtains the source code from version control, runs the commands of the application's build tool, and finally pushes the resulting artifact to a Repository Manager.

2. **Build Image** — The CI Agent pulls the Dockerfile and associated files to build the image from version control. The Dockerfile is setup so that the artifact built in the previous step is copied into the image. The resulting image is pushed to DTR. If Docker Content Trust ([https://docs.docker.com/datacenter/ucp/2.0/guides/content-trust/](https://docs.docker.com/datacenter/ucp/2.0/guides/content-trust/)) has been enabled and Notary ([https://docs.docker.com/datacenter/ucp/2.0/guides/content-trust/manage-trusted-repositories/](https://docs.docker.com/datacenter/ucp/2.0/guides/content-trust/manage-trusted-repositories/)) has been installed, then the image is signed with the CI/CD signature.

3. **Deploy Application** — The CI Agent can pull a runtime configuration from version control (e.g. `docker-compose.yml` + environment-specific configuration files) and use them to deploy the application on UCP.
4. **Test Application** — The CI Agent deploys a test container to test the application deployed in the previous step. If all of the tests pass, then the image can be signed with a QA signature by pulling and pushing the image to DTR. This push triggers the Operations workflow.

**Tip:** The CI Agent can also be Dockerized, however, since it runs Docker commands, it needs access to the host's Docker Engine. This can be done by mounting the host's Docker socket, for example:

```bash
docker run --rm -it --name ciagent 
-v /var/run/docker.sock:/var/run/docker.sock 
  ciagent:1
```

**Tip:** Use Docker images as build caches. When applications are built in a container, often there is a need to download dependencies to compile the application. An enterprise cache or proxy such as Nexus is also used to speed this up. To speed the build up even further you can create a “build base image” with pre-loaded common dependencies from which the build container can inherit. This avoids mounting shared volumes to cache the dependencies between build containers.

---

**CI/CD Environment and Tools**

The nodes of the CI/CD environment where Docker is used to build applications or images should have Docker Engine installed. The nodes can be labeled "build" to create a separate cluster.

There are many CI/CD software systems available (Jenkins, Visual Studio, Team City, etc). Most of the leading systems have some support for Docker through plugins or add-ons. However, to ensure the most flexibility in creating CI/CD workflows, it is recommended that you use the native Docker CLI or rest API for building images or deploying containers/services.

**Operations Workflow**

The Operations workflow usually consists of two parts. It starts at the beginning of the entire development pipeline creating base images for development teams to use, and it ends with pulling and deploying the production ready images from the developer teams. The workflow for creating base images is the same as the developer workflow, so it is not shown here. However, the following diagram illustrates a typical Operations workflow for deploying images in production:
A typical Operations workflow follows these steps:

1. **Deploy Application** — The deployment to production can be triggered automatically via a change to version control for the application or it can be triggered by operations. It can also be executed manually or done by a CI/CD agent. A tag of the deployment configuration files specific to the production environment is pulled from version control. This includes a `docker-compose` file or scripts which deploy the services as well as configuration files. Secrets such as passwords or certificates that are specific to production environments should be added or updated. Starting with Docker 17.03 (and Docker Engine 1.13), Docker has native secrets management. The CD Agent can then deploy the production topology in UCP.

2. **Test Application** — The CD Agent deploys a test container to test the application deployed in the previous step. If all of the tests pass, then the application is ready to handle production load.

3. **Balance the Load** — Depending on the deployment pattern (Big Bang, Rolling, Canary, Blue Green, etc.), an external load balancer, DNS server, or router is reconfigured to send all or part of the requests to the newly deployed application. The older version of the application can remained deployed in case of the need to rollback. Once the new application is deemed stable, the older version can be removed.

### Enterprise Base Images

The Operations team will usually build and maintain “base images.” They typically contain the OS, middleware, and tooling to enforce enterprise policies. They might also contain any enterprise credentials used to access repositories or license servers. The development teams can then inherit from these base images by using the keyword `FROM` in their Dockerfile and then adding their application specific components, applications, and configuration to their own application images.
**Tip:** Squash function. Since the base images do not change that often and are widely used within an organization, minimizing their size is very important. You can use `docker build --squash -t <image>` to create only one layer and optimize the size of the image. You will lose the ability to modify, so this is recommended for base images and not necessarily for application images which change often.

**Production Environment**


**Summary**

This document discusses the Docker development pipeline, integration with existing systems, and also covers the specific developer, CI/CD, and operations workflows and environments. Follow these best practices to create an effective enterprise development pipeline.