Authored by: Andrew Hromis (/author/ahromis)

8 0 Share PDF (https://success.docker.com/api/articles/logging-best-practices/pdf)

Docker Reference Architecture: Docker Logging Design and Best Practices

EE LOGGING DEPLOY MANAGEMENT EE-17.03.0-EE-5 EE-17.06.1-EE

Introduction

Traditionally, designing and implementing centralized logging is an after-thought. It is not until problems arise that priorities shift to a centralized logging solution to query, view, and analyze the logs so the root-cause of the problem can be found. However, in the container era, when designing a Containers-as-a-Service (CaaS) platform with Docker Enterprise Edition (Docker EE), it is critical to prioritize centralized logging. As the number of micro-services deployed in containers increases, the amount of data produced by them in the form of logs (or events) exponentially increases.

What You Will Learn

This reference architecture provides an overview of how Docker logging works, explains the two main categories of Docker logs, and then discusses Docker logging best practices.

Understanding Docker Logging

Before diving into design considerations, it's important to start with the basics of Docker logging.

Docker supports different logging drivers used to store and/or stream **container** `stdout` and `stderr` logs of the main container process (`pid 1`). By default, Docker uses the `json-file` logging driver, but it can be configured to use many other drivers (https://docs.docker.com/engine/admin/logging/overview/#supported-logging-drivers) by setting the value of `log-driver` in `/etc/docker/daemon.json` followed by restarting the Docker daemon to reload its configuration.

The logging driver settings apply to **ALL** containers launched after reconfiguring the daemon (restarting existing containers after reconfiguring the logging driver does not result in containers using the updated config). To override the default container logging driver run the container with `--log-driver` and `--log-opt` options. Swarm-mode services, on the other hand, can be updated to use a different logging driver on the go by using `docker service update --log-driver <DRIVER_NAME> --log-opt <LIST OF OPTIONS> <SERVICE NAME>` .

What about Docker Engine logs? These logs are typically handled by the default system manager logger. Most of the modern distros (CentOS 7, RHEL 7, Ubuntu 16, etc.) use `systemd` , which uses `journald` for logging and `journalctl` for accessing the logs. To access the Engine logs use `journalctl -u docker.service` .

Docker Logs Categories and Sources

Now that the basics of Docker logging have been covered, this section explains their **categories** and **sources**.

Docker logs typically fall into one of two categories: **Infrastructure Management** or **Application** logs. Most logs naturally fall into these categories based on the roles of who needs access to the logs.

- Operators are mostly concerned with the stability of the platform as well as the availability of the services.
- Developers are more concerned with their application code and how their service is performing.

In order to have a self-service platform, both operators and developers should have access to the logs they need in order to perform their role. DevOps practices suggest that there is an overall, shared responsibility when it comes to service availability and performance. However, everyone shouldn't need access to every log on the platform. For instance, developers should only need access to the logs for their services and the integration points. Operators are more concerned with Docker daemon logs, UCP and DTR availability, as well as service availability. There is a bit of overlap since developers and operators both should be aware of service availability. Having access to the logs that each role needs allows for simpler troubleshooting when an issues occurs and a decreased Mean Time To Resolve (MTTR).

Infrastructure Management Logs

The infrastructure management logs include the logs of the Docker Engine, containers running UCP or DTR, and any containerized infrastructure services that are deployed (think containerized monitoring agents).

Docker Engine Logs

As previously mentioned, Docker Engine logs are captured by the OS's system manager by default. These logs can be sent to a centralized logging server.

UCP and DTR System Logs

UCP and DTR are deployed as Docker containers. All their logs are captured in the containers' `STDOUT` / `STDERR` . The default logging driver for Docker Engine captures these logs.

UCP can be configured to use remote syslog logging. This can be done post-installation from the UCP UI for all of its containers.

> Note: It is recommended that the Docker Engine default logging driver be configured **before** installing UCP and DTR so that their logs are captured by the chosen logging driver. This is due to the inability to change a container's logging driver once it had been created. The only exception to this is `ucp-agent` , which is a component of UCP that gets deployed as a Swarm service.

Infrastructure Services

Infrastructure operation teams deploy containerized infrastructure services used for various infrastructure operations such as monitoring, auditing, reporting, config deployment, etc. These services also produce important logs that need to be captured. Typically, their logs are limited to the `STDOUT` / `STDERR` of their containers, so they are also captured by the Docker Engine default logging driver. If not, they need to be handled separately.

Application Logs

Application-produced logs can be a combination of custom application logs and the `STDOUT` / `STDERR` logs of the main process of the application. As described earlier, the `STDOUT` / `STDERR` logs of all containers are captured by the Docker Engine default logging driver. So, no need to do any custom configuration to capture them. If the application has custom logging (e.g. writes logs to `/var/log/myapp.log` within the container), it's important to take that into consideration.

Docker Logging Design Considerations

Understanding the types of Docker logs is important. It is also important to define which entities are best suited to consume and own them.

Categorizing the Docker Logs

Mainly, there are two categories: infrastructure logs and application logs.

Defining the Organizational Ownership

Based on the organization's structure and policies, decide if these categories have a direct mapping to existing teams. If they do not, then it is important to define the right organization or team responsible for these log categories:

Category	Team
System and Management Logs	Infrastructure Operations
Application Logs	Application Operations

If the organization is part of a larger organization, these categories may be too broad. Sub-divide them into more specific ownership teams:

Category	Team
Docker Engine Logs	Infrastructure Operations
Infrastructure Services	Infrastructure Operations
UCP and DTR Logs	UCP/DTR Operations
Application A Logs	Application A Operations
Application B Logs	Application B Operations

Some organizations don't distinguish between infrastructure and application operations, so they might combine the two categories and have a single operations team own them.

Category	Team
System and Management Logs	Infrastructure Operations
Application Logs	Infrastructure Operations

Pick the right model to clearly define the appropriate ownership for each type of log, resulting in decreased mean time to resolve (MTTR). Once organizational ownership has been determined for the type of logs, it is time to start investigating the right logging solution for deployment.

Picking a Logging Infrastructure

Docker can easily integrate with existing logging tools and solutions. Most of the major logging utilities in the logging ecosystem have developed Docker logging or provided proper documentation to integrate with Docker.

Pick the logging solution that:

1. Allows for the implementation of the organizational ownership model defined in the previous section. For example, some organizations may choose to send all logs to a single logging infrastructure and then provide the right level of access to the functional teams.
2. The organization is most familiar with! This is a must!
3. Has Docker integration: pre-configured dashboards, stable Docker plugin, proper documentation, etc.

Docker Enterprise Edition Logs

With Docker Enterprise Edition, it's a good idea to store all of the container logs for historical and system maintenance purposes. It is recommended that you collect the output of some of the containers in an indexable way, mostly for policy reasons and to quickly understand cluster events. In the following sections we'll break down some Docker EE components and certain logs that could be useful to an organization.

UCP

Container Name	Information in Logs
ucp-controller	UCP API logs, users, and logins
ucp-auth-api	Centralized service for identity and authentication used by UCP and DTR
ucp-auth-store	Stores authentication configurations as well as data for users, organizations, and teams
ucp-auth-worker	Performs scheduled LDAP synchronizations and cleans authentication and authorization data
ucp-client-root-ca	Certificate authority to sign client bundles
ucp-cluster-root-ca	Certificate authority to sign client bundles
ucp-metrics	Used for metrics gathering
ucp-kv	etcd service used to store the UCP configurations
ucp-proxy	TLS proxy that allows secure access from the local Docker Engine to UCP components
ucp-swarm-manager	Manager container for Docker Swarm (classic), provides backwards compatibility

Notes:

1. **ucp-controller** - This container logs all login attempts and general usage of the cluster. For auditing purposes, this contains the most important logs in terms of logging usage of the cluster.
2. **ucp-kv** - This container is good to monitor to make sure quorum is not lost on the cluster. If the quorum is lost, it is good practice to setup an alert.

DTR

Name	Information in Logs
dtr-api-	Executes the DTR business logic and serves the DTR web application and API
dtr-garant-	Manages DTR authentication
dtr-jobrunner-	Runs cleanup jobs in the background
dtr-nautilusstore-	Stores security scanning data
dtr-nginx-	Receives HTTP and HTTPS requests and proxies them to other DTR components, by default listens to ports 80 and 443 of the host
dtr-notary-server-	Receives, validates, and serves content trust metadata, and is consulted when pushing or pulling to DTR with Content Trust enabled
dtr-notary-signer-	Performs server-side timestamp and snapshot signing for content trust metadata
dtr-registry-	Implements the functionality for pulling and pushing Docker images, also handles how images are stored
dtr-rethinkdb-	Stores persisting repository metadata in a database

Some notable logs to possibly setup regular expressions on:

1. **dtr-registry-** - Parsing this container will show the client IP's, the user, and general usage of the cluster.
2. **dtr-nginx-** - This container logs all pushes, pulls, and API calls to the cluster.
3. **dtr-rethinkdb-** - The logs in this container contain information about the quorum state of RethinkDB. This is good to monitor and be alerted on any loss of quorum.

HTTP Routing Mesh

By default the HTTP Routing Mesh doesn't log any requests to `STDOUT`. To log from HRM run the following command:

```
docker service update --env-add DEBUG=1 ucp-hrm
```

Logs will then output from the HTTP Routing Mesh. If a log driver is configured at the Engine level, the log output follows the configuration at the Engine level.

Application Log Drivers

Docker has several available logging drivers that can be used for the management of application logs. Check the Docker docs (https://docs.docker.com/engine/admin/logging/overview/#supported-logging-drivers) for the complete list as well as detailed information on how to use them.

As a general rule, if you already have logging infrastructure in place, then you should use the logging driver for that existing infrastructure. If you don't have an existing logging system in place, there's a few areas worth highlighting advantages and disadvantages.

Driver	Advantages	Disadvantages
none	Ultra-secure, since nothing gets logged	Much harder to troubleshoot issues with no logs

Driver	Advantages	Disadvantages
json-file (https://docs.docker.com/engine/admin/logging/json-file/)	The default, `docker logs` works with it, supports tags	Logs reside locally and not aggregated, logs can fill up local disk if no restrictions in place
syslog (https://docs.docker.com/engine/admin/logging/syslog/)	Most machines come with syslog, only driver that supports TLS for encrypted log shipping, supports tags	Needs to be set up as highly available (HA) or else there can be issues on container start if it's not available
journald (https://docs.docker.com/engine/admin/logging/journald/)	`docker logs` also works with this driver, since it logs locally - log aggregator can be down without impact, this also collects Docker daemon logs	Since journal logs are in binary format, extra steps need to be taken to ship them off to the log collector, no tag support
gelf (https://docs.docker.com/engine/admin/logging/gelf/)	Provides indexable fields by defaults (container id, host, container name, etc.), tag support	Only supports the UDP protocol, no `docker logs` support
fluentd (https://docs.docker.com/engine/admin/logging/fluentd/)	Provides `container_name` and `container_id` fields by default, fluentd supports multiple outputs,	No `docker logs` support for local logs
awslogs (https://docs.docker.com/engine/admin/logging/awslogs/)	Easy integration when using Amazon Web Services, less infrastructure to maintain, tag support	Not the most ideal for hybrid cloud configurations or on-premise installations, no `docker logs` support
splunk (https://docs.docker.com/engine/admin/logging/splunk/)	Easy integration with Splunk, TLS support, highly configurable, tag support, additional metrics	Splunk needs to be highly available or possible issues on container start, no `docker logs` support
etwlogs (https://docs.docker.com/engine/admin/logging/etwlogs/)	Common framework for logging on Windows, default indexable values	Only works on Windows, those logs have to be shipped from Windows machines to a log aggregator with a different utility
gcplogs (https://docs.docker.com/engine/admin/logging/gcplogs/)	Simple integration with Google Compute, less infrastructure to maintain, tag support	Not the most ideal for hybrid cloud configurations or on-premise installations, no `docker logs` support

Application Log Driver Considerations

Consider the following when selecting application log drivers:

- If log data is highly sensitive, then **syslog** and **splunk** are good options since they can be configured to use TLS for transporting logs.
- The **journald** log driver is great for retaining the usage of `docker logs` as well as logging Docker daemon logs. This driver allows for easier troubleshooting and log portability at the same time. Another advantage of this driver is that logs will write first locally, so that there is less reliance on logging infrastructure.
- If the Docker EE cluster exist solely on a single cloud provider, then **awslogs** or **gcplogs** can be used.
- If there's an existing Splunk installation, then use the **splunk** log driver.
- The **gelf** and **fluentd** log drivers are a good choice if there's a NoSQL database somewhere in the environment where the logs can be stored.
- For development or test environments, using **json-file** or **journald** could be useful where it's more useful to view a log stream rather than index and search the logs. (If **json-file** is used consider passing the `max-size` and `max-file` options so that logs won't fill up the filesystem.)

Logging Driver Setup Walkthrough

To implement system-wide logging, creating an entry in `/etc/docker/daemon.json` . For example, use the following to enable the `gelf` output plugin:

```
{
    "log-driver": "gelf",
    "log-opts": {
     "gelf-address": "udp://1.2.3.4:12201",
     "tag":"{{.ImageName}}/{{.Name}}/{{.ID}}"
    }
}
```

And then restart the Docker daemon. All of the logging drivers can be configured in a similar way, by using the `/etc/docker/daemon.json` file. In the previous example using the `gelf` log driver, the `tag` field sets additional data that can be searched and indexed when logs are collected. Please refer to the documentation for each of the logging drivers to see what additional fields can be set from the log driver.

Setting logs using the `/etc/docker/daemon.json` file will set the default logging behavior on a per-node basis. This can be overwritten on a per-service or a per-container level. Overwriting the default logging behavior can be useful for troubleshooting so that the logs can be viewed in real-time.

If a service is created on a system where the `daemon.json` file is configured to use the `gelf` log driver, then *all* container logs running on that host will go to where the `gelf-address` config is set.

If a different logging driver is preferred, for instance to view a log stream from the `stdout` of the container, then it's possible to override the default logging behavior ad-hoc.

```
docker service create \
      --log-driver json-file --log-opt max-size=10m \
      nginx:alpine
```

This can then be coupled with Docker service logs to more readily identify issues with the service.

Docker Service Logs

`docker service logs` was introduced in version 17.05 of and version 17.06 of Docker EE. It provides a multiplexed stream of logs when a service has multiple replica tasks. By entering in `docker service logs <service_id>`, the logs show the originating task name in the first column and then real-time logs of each replica in the right column. For example:

```
$ docker service create -d --name ping --replicas=3 alpine:latest ping 8.8.8.8
5x3enwyyr1re3hg1u2nogs40z

$ docker service logs ping
ping.2.n0bg40kksu8e@m00    | 64 bytes from 8.8.8.8: seq=43 ttl=43 time=24.791 ms
ping.3.pofxdol20p51@w01    | 64 bytes from 8.8.8.8: seq=44 ttl=43 time=34.161 ms
ping.1.o07dvxfx2ou2@w00    | 64 bytes from 8.8.8.8: seq=44 ttl=43 time=30.111 ms
ping.2.n0bg40kksu8e@m00    | 64 bytes from 8.8.8.8: seq=44 ttl=43 time=25.276 ms
ping.3.pofxdol20p51@w01    | 64 bytes from 8.8.8.8: seq=45 ttl=43 time=24.239 ms
ping.1.o07dvxfx2ou2@w00    | 64 bytes from 8.8.8.8: seq=45 ttl=43 time=26.403 ms
```

This command is useful when trying to view the log output of a service that contains multiple replicas. Viewing the logs in real time, streamed across multiple replicas allows for instant understanding and troubleshooting of service issues across the entire cluster.

ELK Setup

Please refer to the swarm-elk repository (https://github.com/ahromis/swarm-elk) for information on how to send logs to ELK on a Docker Swarm. This repo contains a Docker Compose file that sets up a complete ELK stack. The repository is a good starting point for experimenting with Docker logging with ELK, but also consider high availability and a queuing system before it's used in a production manner.

Splunk Setup

Splunk is another popular logging utility. To set up Splunk follow the steps using the Docker Compose files available in this swarm-splunk (https://github.com/ahromis/swarm-splunk) repository.

Each Splunk forwarder connects to the local Docker socket, so it doesn't need additional log driver configuration at the daemon level. Connecting to the local socket also allows Splunk to pull out Docker container statistics in addition to logs.

Modernize Traditional Applications

There's no reason to think that logging using containers is only meant for modern applications. It's possible to modernize traditional applications and still have the added benefit of modernized logging without changing any application code.

Ideally, applications log to `stdout` / `stderr`, and Docker sends those logs to the configured logging destination. Sometimes certain applications are configured to log to multiple locations. How can those logs be captured without having to change any of the application source code?

It's possible to create a Docker volume (or many volumes) directed at where those log files reside within the application. By leveraging Docker templating, it's possible to suffix each volume with the service task ID (an integer). Placing a suffix on the volume name prevents any collisions in regards to logging should multiple service tasks run on the same host. A global service needs to be created that runs a logging agent with directory wildcard support. Finally, additional regex can be setup via the logging utility that turns the source directory of the file, into an indexed value.

The following example shows how this can be accomplished using the official Tomcat (https://store.docker.com/images/tomcat) image. The official Tomcat image logs several files in `/usr/local/tomcat/logs`, much like most Java applications would do. In that path, files such as `catalina.2017-07-06.log`, `host-manager.2017-07-06.log`, `localhost.2017-07-06.log`, `localhost_access_log.2017-07-06.txt`, and `manager.2017-07-06.log` can be found.

1. Create a global service for the logging utility that mounts `/var/lib/docker/volumes:/logs/volumes`.
2. Create a logging rule for the logging agent that logs using a rule similar to this generic example: `"/log/volumes/*/_data/*.log"`.
3. Launch a service using go based templating on the volumes:

   When launching the service, use these parameters:

   ```
   docker service create \
   -d \
   --name prod-tomcat \
   --label version=1.5 \
   --label environment=prod \
   --mount type=volume,src="{{.Task.Name}}",dst=/usr/local/tomcat/logs \
   --replicas 3 \
   tomcat:latest
   ```

   If both replicas schedule on the same node, then two volumes containing the logs will be create on the host `prod-tomcat.1.oro7h0e5yow69p9yumaetor3l` and `prod-tomcat.2.ez0jpuqe2mkl6ppqnuffxqagl`.
4. As long as the logging agent supports wildcards and handles any log rotation by checking the inode (not the file), then the logs should be collected.
5. If the application logs to multiple locations, then try to symlink the logs to a single directory or add a descriptive name to the volume. If a descriptive name is added to the volume name, then any sort of extraction will need to be updated to accommodate that change. (i.e. with a `grok`)

6. Most loggers should collect the file path as well as the log contents. By turning the volumes where the log files reside into indexable fields it's possible to search and aggregate information from these types of applications. Here is an example that uses a `grok` pattern and creates two new indexable fields, `CONTAINER_NAME` and `FILENAME`.

   ```
   match => { "path" => "/log/volumes/%{DATA:CONTAINER_NAME}/_data/%{GREEDYDATA:FILE_NAME}" }
   ```

7. The `CONTAINER_NAME` will match the output of the `stdout` stream from the container, making it easy to filter based on the container's logs.

> More information to a repository with a working example can be found in the swarm-elk repo (https://github.com/ahromis/swarm-elk/tree/master/logstash).

Windows Logging

The ETW logging driver forwards container logs as ETW events. ETW stands for Event Tracing in Windows and is the common framework for tracing applications in Windows. Each ETW event contains a message with both the log and its context information. A client can then create an ETW listener to listen to these events and forward them to a location where the logs can be collected and analyzed.

To log using the ETW driver on Windows, create a service or run a container with the flag `--log-driver=etwlogs`.

Here's is an example event message:

```
container_name: backstabbing_spence,
image_name: windowsservercore,
container_id: f14bb55aa862d7596b03a33251c1be7dbbec8056bbdead1da8ec5ecebbe29731,
image_id: sha256:2f9e19bd998d3565b4f345ac9aaf6e3fc555406239a4fb1b1ba879673713824b,
source: stdout,
log: Hello world!
```

Conclusion

Docker provides many options when it comes to logging, and it's helpful to have a logging strategy when adopting the platform. For most systems, leaving the log data on the host isn't adequate. Being able to index, search, and have a self-service platform allows for a smoother experience for both operators and developers.

Docker for Windows Server (https://www.docker.com/docker-windows-server)

Docker for Debian (https://www.docker.com/docker-debian)

Docker for Fedora® (https://www.docker.com/docker-fedora)

Docker for Oracle Linux (https://www.docker.com/docker-oracle-linux)

Docker for RHEL (https://www.docker.com/docker-red-hat-enterprise-linux-rhel)

Docker for SLES (https://www.docker.com/docker-suse-linux-enterprise-server-sles)

Docker for Ubuntu (https://www.docker.com/docker-ubuntu)

Documentation (https://docs.docker.com/)

Blog (https://blog.docker.com/)

RSS Feed (https://blog.docker.com/feed/)

Training (https://training.docker.com/)

Knowledge Base (https://success.docker.com/kbase)

Resources (https://www.docker.com/products/resources)

Community (https://www.docker.com/docker-community)

Open Source (https://www.docker.com/technologies/overview)

Events (https://www.docker.com/community/events)

Forums (https://forums.docker.com/)

Docker Captains (https://www.docker.com/community/docker-captains)

Scholarships (https://www.docker.com/community-partnerships)

Community News (https://blog.docker.com/curated/)

Status (http://status.docker.com/) Security (https://www.docker.com/docker-security) Legal (https://www.docker.com/legal) Contact (https://www.docker.com/company/contact)

(http: (http: (https (https (https (https (http: (http: