Guide to Building Docker Images

Document Level Classification

200

Introduction

Docker images are the primary means to deliver code in most modern deployment systems. They are used for deployments into Substrate Kubernetes as well as Substrate Managed Services.

This document describes best practices for building Docker images for Managed Services, though this guidance is also applicable to workloads running on Substrate Kubernetes.

We presume you are comfortable reading Docker's reference documentation and have already built at least one Docker image. If you are completely new to Docker we recommend you start with the Docker overview and come back to this document when you have built at least one container.

Readiness Checklist

To help you follow the document below, we've included a short readiness checklist for you to follow:

1. [ ] I'm using ENV for environment variables
2. [ ] I'm using EXPOSE for networking
3. [ ] I'm using CMD to start my service
4. [ ] I'm using health and readiness probes to monitor my service's health

Environment Variables

Most services need some kind of configuration to start, whether database connection information or an API credentials. We recommend you provide these to your service via environment variables, and that you include your default environment variables in your Docker image. You can do so using the ENV directive in your Dockerfile.

This example shows using ENV to specify the service's default database password.

```
ENV DATABASE_ENDPOINT=127.0.0.1:3306
ENV DATABASE_USERNAME=starmap
```

```
ENV DATABASE_PASSWORD=ilikecookies
ENV DATABASE_CONNECTION_STRING=""
```

Adding these ENV directives to your Dockerfile will automatically populate the Environment configuration in the Managed Services UI.

Make sure your default values are valid, since these will be used if you don't specify anything else. You can use `ENV KEY=""` to specify an empty default value.

See the Dockerfile ENV reference for more details.

Do not put production secrets into your Dockerfile – use placeholder values instead. Environment variables in Docker images are not secret and are accessible to anyone with access to Epic's network. Use the Environment configuration in the Managed Services UI to encrypt secret configuration such as database credentials and API keys.

Inbound Networking

Many services listening on a network port and accepting connections from people or other programs. When you setup such a service, you will need to indicate which ports and protocols it listens on in order for a load balancer to direct traffic to your service.

Docker provides the EXPOSE directive in the Dockerfile to facilitate this process.

This example indicates that the service is listening on 8080 (the load balancer may listen on a different port):

```
EXPOSE 8080
```

Adding these EXPOSE directives in your Dockerfile will automatically populate the Ports configuration in the Managed Services UI.

See the Dockerfile EXPOSE reference for more details.

Default Command

Docker allows you to specify the default command used when running your container via the CMD directive. You should configure CMD in your Dockerfile to include your main service binary and any command line flags or arguments it needs to start successfully. Keep in mind that the environment will come from ENV directives, so you don't need to specify these here.

This example runs the `/starmap-api` binary and invokes the `server` subcomand:

```
CMD ["/starmap-api","server"]
```

CMD **must** be specified in order for your service to start on Managed Services.

See the Dockerfile CMD reference for more details.

Do not use `ENTRYPOINT` in your Dockerfile. `ENTRYPOINT` blocks access to your /healthz and `/readyz` probes. See below for more info.

Health and Readiness

Any large system is bound to be broken in some way, whether it's a bad disk, disrupted network, or bug in service code. Health and readiness checks are an important part of detecting and fixing parts of our service fleet that have become broken or degraded.

We recommend that all services implement health and readiness probes.

Fortunately, these are easy. When deploying to Epic Cloud Managed Services you have three options for health and readiness checks.

1. Exit status (*very* simple, for *very* simple services)
2. HTTP Probe
3. Check scripts

We'll cover these in detail below.

Readiness Probes

When your service first starts it may need time to initialize before it is ready to serve traffic. For example, it may take time to JIT, warm caches, load data into memory, or establish connections to other systems.

With a readiness probe configured, the platform will continuously check your service to see when it is ready to serve traffic, and only after the service indicates that it's ready will it receive traffic from a load balancer.

Health Probes

Your service's health may degrade over time. For example, memory leaks, stuck threads, or dropped network connections may eventually pile up, causing your service to degrade or stop responding.

Implementing health probes allows the platform to identify individual service instances that are failing or degraded and replace them with new, healthy instances.

The platform continuously runs check scripts on each instance of your service to determine whether it is still healthy. When the platform detects a series of failures on the health probe, your workload will be restarted.

Exit Status

Exit status is the most rudimentary means to check whether your service is healthy. In this mode, the platform checks whether your process is still

running or has exited with an error code. When your program exits with an error code the platform reports a failure and restarts your process.

For very simple services like background tasks and cron jobs, exit status checks are adequate. However, most services will benefit from more sophisticated checks.

HTTP Probe

Using an HTTP probe, the platform will periodically call your service at the `/readyz` or `/healthz` endpoints to check whether it is ready to receive traffic or is encountering some kind of problem. A 200 response indicates that the service is ready or healthy. Any other response indicates an error state. When the platform detects a series of failures on this probe your workload will be restarted.

The benefit of using HTTP probes is that they are generally easy to implement. You can write them in your service code and they can check all of the things that your service needs, like database connections, credentials, access to network resources, etc. They can also provide diagnostic information to you if you want to see what your service is doing.

Check Scripts

Using a check script, the platform will periodically run a program inside your container at the `/readyz` or `/healthz` paths and record the exit status. A check script written in this way can be a as simple as a bash script or curl call, or an entire program written to analyze your service externally. For example, you could call JXM hooks, assess total memory consumption for your process, or analyze stats over time.

Exit status 0 indicates success. Any other exit code indicates failure. When the platform detects a series of failures on this probe your workload will be restarted.

Check scripts provide unlimited flexibility and are a good way to implement a check for a headless service like a video encoder that doesn't

have an HTTP endpoint at all. You can also use check scripts to write a probe that is more sophisticated than the basic HTTP probe provided by the platform.
