Docker - Ruby on Rails

Overview

Building Dockerfiles and docker-compose files for Ruby on Rails projects can be challenging due to the differences in typical workflows and practices between the Ruby on Rails and container based development.

Ruby Version

Since there many versions of ruby used throughtout different projects, to create a Dockerfile quickly based of a pre-existing image from Dockerhub, the Ruby version must be available. While base images of most versions are available, some may not be.

If the version is not available rvm can be used to change versions using:

'curl get.rvm.io | bash'

However due to how rvm works it needs to be run in a login shell. Since the shell docker runs when started is not a login shell, commands which require rvm to manage ruby must be run in:

RUN bash -l -c "<ruby-command>"

Bundler

In rails projects, bundler manages gem dependencies, ensuring all which are required for each environment and installed. The gems for each of the project's environment are listed in the Gemfile. Since the gems can be source of a large amount of data and processing, to optmise the container image, they should be added in early so they can be cached. To do this, copy the file in early and run bundle:

COPY Gemfile Gemfile.lock ./

RUN bundle install

When adding new gems to a project, if the original gemfile is not large it is possible to add directly to that knowing that it will trigger a build of all artifacts added after it. Otherwise, if the project is far into development and the gemfile established, it is also possible to create a Gemfile.tip file, and add it in later in the build.

Rails Environment

Common practice in Docker is to the use the same container images for dev, testing, staging, and production. However, Rails does not work in this way since it tends to have different artifacts for each environment. Changing the environment changes:

* the set of gems
* how static assets are processed
* which databases are used

While it works well for Ruby development, it is not the standard Docker model.

Asset Pipeline

In development, assets such as JavaScript, CSS, and images are processed on the fly by the application server. However, in production assets are pre-compiled to be served by a static server such as a CDN or the web server.

When compiling the assets for a production image, generally the whole rails application is required which can cause issues for creating a clean image. There are a few ways to handle asset compilation, the two typical ways have the application server and web server in separate containers:

* Assets are pre-compiled at built-time by the app server container, then transferred to the webserver container as required. Issues can occur if access to the production database is required and it is not running and available to the build (this can be solved by mocking databases).
* Assets are pre-compiled at build-time, which achives clean seperation and removes the issue of the database not being available. However, it does mean there will be slight downtime where the assets will not be available when the app and web container starts.

When deploying to an CDN assets server, it is possible to pass through the assets server path as a user defined arg during build. This allows the rails application to know where the assets are being served from.

Migrations

Since there is no central machine with containers, in a traditional system multiple container deploy would lead to mutliple db:migrate commands being run at the same time. A way to handle this is to use an entrypoint which:

* waits for the database server to be up and running
* check if the database exists and if not create it
* check if migrations are pending and if so run them
* if running pending migrations failed, try again and it will skip if they are not required