One Way To: Containerize a Ruby on Rails Application

One of the many ways to define and build a Ruby on Rails v6 application container image for production deployment.

### Introduction

While Ruby on Rails framework was not originally designed to be containerized, with it emerging before containerization becoming mainstream, with a little adjustment to its configuration and careful application architecture, it can be packaged up quite easily. Here one of the ways I have found to containerize a basic rails application for production (containerization for development and test to follow).

## Why Containerize

There are many reasons for one to want to containerize a rails application, most of which tie into the benefits of containers. Below are a few of the pros and cons:

#### Pros

- \*\*Scalability\*\* - Compared to traditional VM environments, containers are very quick to spin up and stop, allowing for fast on-demand scaling.

- \*\*Portability\*\* - Applications in the container will run the same regardless of where they are deployed. This consistent reproducibility allows for easy deployment on a range of different machines.

- \*\*Resource efficiency\*\* - Containerized applications reduce wasted resources. Each container only holds its application, while sharing the host's kernel. The containers can then be deployed efficiently with resource limits, to maximise the host instance's utilization.

- \*\*Workflow\*\* - The whole workflow from test and development to deployment can be performed in a containerized environment. The environmental consistency across the workflow provides fast development setup on any machine, prevents cross-platform bugs, and identifies all dependencies.

#### Cons

- \*\*Persistent storage\*\* - By design, local storage in containers is temporary, and is lost when the container is stopped. Coupled with the fact containers are ephemeral, persistent storage requires managing through carefully managed mounted volumes or backing services.

- \*\*Performance\*\* - Not quite as fast as bare-metal due to overhead from various aspects of container architecture, such as overlay networking. However, performance is normally offset by resource efficiency.

Overall if you're building a stateless application with the need to scale efficiently and effectively, containerizing it could serve you well.

## How

### Prerequisites

This post assumes you already have a simple Ruby on Rails application which you want to encapsulate in a container. I created a sample containerized Ruby on Rails application, which I will refer to throughout this post, the complete codebase can be found here: [https://github.com/cpcwood/containerized\_blog](https://github.com/cpcwood/containerized\_blog)

### Architecture

Since containers are required to be ephemeral (short-lived), the application must be stateless and have no reliance on local storage for persistent or shared data. In Rails, this means designing the application to:

- store client session data in cookies or a backing service

- use backing services, such as external databases and AWS S3, for persistent data storage

- logging to stdout and managing centralized logging externally

A common architectural starting point for a containerized application is the [12 factor app]( https://12factor.net/), which if followed will help cover many of the design quirks of applications running in containers.

### Configuration

As described in the [12 factor app]( https://12factor.net/), the application configuration is stored in environment variables. Therefore, go through the application and convert user configuration be extracted from the environment variables.

Update ```config/database.yml```:

```yaml

default: &default

adapter: postgresql

encoding: unicode

host: <%= ENV.fetch("DB\_HOST") { 'localhost' } %>

pool: <%= ENV.fetch("RAILS\_MAX\_THREADS") { 5 } %>

username: <%= ENV["DB\_USERNAME"] %>

password: <%= ENV["DB\_PASSWORD"] %>

port: 5432

development:

<<: \*default

database: <%= ENV.fetch("DB\_NAME") { 'containerized\_blog\_development' } %>

test:

<<: \*default

database: <%= ENV.fetch("DB\_NAME") { 'containerized\_blog\_test' } %>

production:

<<: \*default

database: <%= ENV.fetch("DB\_NAME") { 'containerized\_blog\_production' } %>

```

Update ```config/storage.yml```:

```yaml

# ...

amazon:

service: S3

access\_key\_id: <%= ENV.fetch('AWS\_ACCESS\_KEY\_ID') { '' } %>

secret\_access\_key: <%= ENV.fetch('AWS\_SECRET\_ACCESS\_KEY') { '' } %>

region: <%= ENV.fetch('AWS\_REGION') { '' } %>

bucket: <%= ENV.fetch('AWS\_BUCKET') { '' } %>

```

Make sure to keep a note of the environment variables required to configure the application. To do this you can create a new directory and template config file, for example:

```sh

mkdir config/env\_vars

vim config/env\_vars/.env.template

```

```sh

# config/env\_vars/.env.template

# Application Server Settings

RAILS\_MAX\_THREADS=16

RAILS\_MIN\_THREADS=1

PORT=5000

RAILS\_ENV=<environment>

RAILS\_LOG\_TO\_STDOUT=true

RAILS\_SERVE\_STATIC\_FILES=true

# PSQL Database Credentials

DB\_USERNAME=<your-psql-username>

DB\_PASSWORD=<your-psql-password>

DB\_HOST=<your-psql-host>

DB\_NAME=<your-database-name>

# AWS Credentials

AWS\_ACCESS\_KEY\_ID=<aws-access-id>

AWS\_SECRET\_ACCESS\_KEY=<aws-secret-key>

AWS\_REGION=<aws-bucket-region>

AWS\_BUCKET=<aws-bucket-name>

# Site Settings

SECRET\_KEY\_BASE=<your-secret-key-base>

```

These environment variables can then be added to the container environment during its creation by your container orchestrator.

#### Development and Test Environments

While the 12 factor app methodology disagrees with using config files for separate 'environments', such as ```development```, ```test```, Rails is designed around these environments and sometimes it is not practical to set the environment's environment variables each time you want to run. So, in this case, where we are just containerizing for production, it can be easier to use a gem such as [dotenv]( https://github.com/bkeepers/dotenv) to load environment files.

First, add the ```gem 'dotenv' ``` to your Gemfile, under ```:development``` and ```:test```, and install using ```bundle install```.

Update your ```.gitignore``` file to ignore any env files, ensuring they are not checked into source control, possibly exposing secrets:

```

# …

\*\*/\*.env

```

Create the development and test ```.env``` files:

```sh

# config/env\_vars/dev.env

# insert development config environment variables…

```

```sh

# config/env\_vars/test.env

# insert test config environment variables…

```

Load the ```.env``` files using dotenv at the top of their respective environment files. For example:

```rb

# config/environments/development.rb

require 'dotenv'

Dotenv.load('config/env\_vars/dev.env')

# ...

```

### Docker

[Docker](https://docker.com), is a set of platform as a service (PaaS) products used to define, build, and run Docker container images, and much more. It has quickly become the industry standard for creating containers and is very well documented, so well use it here.

Install Docker, using their [installation guide](https://docs.docker.com/get-docker/) for your machine.

Creating containers using Docker is done in three steps:

1. Define the container image in a Dockerfile

2. Build the container image

3. Run the container image as a container

In this post we will perform the first two steps since once a container image is built, it can be published and be deployed on any machine with container runtime installed, such as Docker, [CRI-O](https://cri-o.io/), and [containerd](https://containerd.io/).

### Dockerfile

A [Dockerfile](https://docs.docker.com/engine/reference/builder/) is a list of commands which when run will assemble a container image.

#### Create the .dockerignore

When copying files into the container image, use a ```.dockerignore``` file (similar to ```.gitignore``` file) to prevent unwanted files from copying and bloating the image size:

Create the ```.dockerignore``` file:

```sh

touch .dockeringore

```

Add patterns for files to be ignored, for example:

```sh

# OS X

.DS\_Store

.AppleDouble

.LSOverride

Icon

.\_\*

.Spotlight-V100

.Trashes

.AppleDB

.AppleDesktop

Network Trash Folder

Temporary Items

.apdisk

# Rails

coverage/

docs/

log/

node\_modules/

public/packs/

public/packs-test/

public/system

spec/

test/

storage/

tmp/

.bundle

.ruby-version

\*.key

\*.log

\*.state

\*.pid

.env

# others

.git

.gitignore

.keep

.vscode

.byebug\_history

.browserslistrc

.rspec

.rubocop.yml

yarn-error.log

logfile

```

#### Create the Dockerfile

Create a new file named ```Dockerfile``` in the root directory of the project:

```sh

touch Dockerfile

```

#### Builder Image

As each instruction in the Dockerfile adds a layer to the image, using a [multi-stage build](https://docs.docker.com/develop/develop-images/multistage-build/) to compile client assets can be an easy way to minimize the end image size.

Image size optimization is important since a smaller image size improves the performance of the build and deployment as less data needs to be pulled to the running container cluster.

Pick a base image which is suitable for the project, ) and use it for the as the builder and main images. For example, ```ruby:2.7.2-alpine```, [official Ruby container images](https://hub.docker.com/\_/ruby) will already be optimised, and alpine is a very lightweight Linux distribution to keep image sizes down.

Then set the standard environment variables for the builder stage and create the working directory. Add temporary ```SECRET\_KEY\_BASE``` to allow for precompiling of assets. Using ```BUNDLE\_PATH``` to keep the gem files local to the application allows for easier copying of the dependencies later:

```dockerfile

# Dockerfile

# Builder Image

FROM ruby:2.7.2-alpine AS builder

ENV RAILS\_ENV=production \

NODE\_ENV=production \

APP\_HOME=/opt/app \

SECRET\_KEY\_BASE=1234567890

ENV BUNDLE\_PATH=$APP\_HOME/vendor/bundle \

BUNDLE\_APP\_CONFIG=$APP\_HOME/vendor/bundle

RUN mkdir -p $APP\_HOME

WORKDIR $APP\_HOME

```

Install application dependencies:

```dockerfile

# Dockerfile

# …

RUN apk add --no-cache \

build-base \

postgresql-dev \

nodejs \

yarn \

git

```

#### Cache Gemfile & package.json

After the initial build of the container image, Docker will cache image layers, only rebuilding them when there has been a change. Since the application Gemfile and package.json do not change very often and add significant time to the build, it is common to copy these into the image early on, to allow the dependencies they install to be cached.

```dockerfile

# Dockerfile

# …

COPY Gemfile\* $APP\_HOME/

RUN bundle config set without development:test:assets && \

bundle install

COPY package.json yarn.lock $APP\_HOME/

RUN yarn install --production=true

```

#### Precompile Assets

Copy the source files and compile the assets, also remove any build artifacts and unrequired dependencies to prevent them from being copied into the main image later on:

```dockerfile

# Dockerfile

# …

COPY . $APP\_HOME

RUN bundle exec rails assets:precompile

RUN rm -rf $APP\_HOME/node\_modules && \

rm -rf $APP\_HOME/app/javascript/packs && \

rm -rf $APP\_HOME/log/\* && \

rm -rf $APP\_HOME/spec && \

rm -rf $APP\_HOME/storage/\* && \

rm -rf $APP\_HOME/tmp/\* && \

rm -rf $APP\_HOME/vendor/bundle/ruby/2.7.0/cache/ && \

find $APP\_HOME/vendor/bundle/ruby/2.7.0/gems/ -name "\*.c" -delete && \

find $APP\_HOME/vendor/bundle/ruby/2.7.0/gems/ -name "\*.o" -delete

```

#### Create the Main Image

Define the main image directly beneath the builder image in a similar fashion. Make sure to only include the application dependencies required for runtime.

```dockerfile

# Dockerfile

# …

# Main Image

FROM ruby:2.7.2-alpine

ENV RAILS\_ENV=production \

NODE\_ENV=production \

APP\_HOME=/opt/app

ENV BUNDLE\_PATH=$APP\_HOME/vendor/bundle \

BUNDLE\_APP\_CONFIG=$APP\_HOME/vendor/bundle

RUN apk add --no-cache \

imagemagick \

postgresql-client \

tzdata && \

cp /usr/share/zoneinfo/Europe/London /etc/localtime && \

echo "Europe/London" > /etc/timezone

RUN mkdir -p $APP\_HOME

WORKDIR $APP\_HOME

```

#### Add Docker User

Run application as unprivileged docker user:

```dockerfile

# Dockerfile

# …

RUN addgroup -S docker && \

adduser -S -G docker docker

USER docker

```

#### Copy Assets from Builder

Copy the compile assets and source code from the builder stage, making sure they are owned by the new docker user:

```dockerfile

# Dockerfile

# …

COPY --chown=docker:docker --from=builder $APP\_HOME $APP\_HOME

```

#### Expose Port

Expose the port the application server will be running on to allow external services and requests to contact the server:

```dockerfile

# Dockerfile

# …

EXPOSE 5000

```

#### Startup Script

Since the container could be created at the beginning of the applications life, or midway through, the database might not be set up or fully migrated to the current image's state.

Create a startup script to first create or migrate the database and then start the application server.

Firstly, create a new rake task to check if the application database exists:

```rb

# lib/tasks/db\_exists.rake

namespace :db do

desc 'Checks to see if the database exists'

task exists: :environment do

Rake::Task['environment'].invoke

ActiveRecord::Base.connection

rescue StandardError

exit 1

else

exit 0

end

end

```

Then create the startup shell script:

```sh

# scripts/container-startup.sh

#!/bin/sh

bundle exec rake db:exists && bundle exec rake db:migrate || bundle exec rake db:setup

bundle exec rails server -b 0.0.0.0 -p 5000

```

Finally, run the startup script upon container creation:

```dockerfile

# Dockerfile

# …

CMD ["./scripts/container-startup.sh"]

```

Take a look at the full Dockerfile defined above [here](https://github.com/cpcwood/containerized\_blog/blob/main/Dockerfile).

### Build the Container Image

Build the container image from the Dockerfile is relatively simple using Docker. Simply run:

```sh

sudo docker build -t <container-name> .

```

Once the container image has been built, it is ready to be published to [DockerHub](https://hub.docker.com/) and or deployed.

### Test Using Docker Compose

While docker can be used to run container images directly, it is not particularly user friendly and repeatable when attempting to run multiple containers, such as the application and database. [Docker compos](https://docs.docker.com/compose/) can be used to configure and run multi-container applications in a repeatable manner. Therefore, it is useful to test your production container images locally, in a pseudo-staging environment.

#### Define the Application

Create a ```docker-compose.yaml``` in the root directory of the project defining the application and database services. For example:

```yaml

version: "3.7"

services:

app:

image: <container-name>

env\_file:

- 'config/env\_vars/.env'

depends\_on:

- postgres

ports:

- '5000:5000'

postgres:

image: postgres:13

environment:

POSTGRES\_USER: cpcwood

POSTGRES\_PASSWORD: password

PGDATA: /var/lib/postgresql/data/pgdata

ports:

- '5432:5432'

volumes:

- /var/db/psql/13:/var/lib/postgresql/data

healthcheck:

test: pg\_isready -U cpcwood -h 127.0.0.1

interval: 5s

```

The above docker-compose file sets up two services, ```app``` and ```postgres```, which run the Ruby on Rails application container and a PostgreSQL database container respectively.

The ```app``` service:

- sets the container image as the name of the container image built earlier

- loads environment variables from the ```config/env\_vars/.env``` file

- won't start until the postgres service is started

- exposes port 5000 to port 5000 of the local machine

The ```postgres``` service:

- uses the [official postgres image](https://hub.docker.com/\_/postgres), which will be pulled from dockerhub if not available locally

- sets the user, password, and data location

- exposes port 5432 to port 5432 of the local machine

- mounts a local volume into the image data location directory, this allows for data to persist between container mounts (ensure the mounted directory ```/var/db/psql/13``` is the correct directory on the local machine for your postgres data/)

- performs a health check to ensure the container stay live

#### Create Environment File

When deploying with a container orchestrator, such as Kubernetes, the config will likely be created in its own object. However, for running the container locally using docker-compose it can be convenient to store the environment variables in a file and load them into the container on creation (make sure not to check it into your source control).

Create the environment file which is to be read and exported into the ```app``` container on creation. For example, create ```config/env\_vars/.env``` from the template ```.env.template``` file created earlier.

Note: the database will not be available on localhost, instead it will be available on the name of the database service: ```postgres```. So, in the example case, set ```DB\_HOST=postgres```

#### Start the Application

Use the docker-compose CLI to start the application:

```sh

sudo docker-compose up

```

Once started the application should now be live on port 5000 on your local machine. Test it by visiting [http://0.0.0.0:5000](http://0.0.0.0:5000) in your browser.

## What's next

#### Deploy

Deploy the application using a container orchestrator, such as [Kubernetes](https://kubernetes.io/) or [Docker Swarm](https://docs.docker.com/engine/swarm/). Alternatively, use a managed system such as AWS's [Amazon ECS](https://aws.amazon.com/ecs/) or Google's [GKE](https://cloud.google.com/kubernetes-engine).

#### Add CI/CD

Building, publishing, and deploying containers can be performed during a continuous integration and continuous delivery (CI/CD) workflow, using services such as [CircleCI](https://circleci.com/) or self-hosted [Jenkins](https://www.jenkins.io/).

With further configuration, CI/CD services can also provide the ability to copy pre-compiled assets to CDN instead of using the backend application server to serve them, lightening up the image size and load, while speeding up response times for clients.

#### Development & Test Environments

Setup containerized development and test environments to allow for a consistent environment throughout the application's workflow, and quick setup on any machine with Docker installed.

#### Explore the Container

You can execute commands in running containers either directly or by running an interactive shell:

- spin up the container using ```sudo docker-compose up```

- get the id of the running container using ```sudo docker containers ls``` and reading the CONTAINER ID

- open shell in container by running the shell executable command in the container: ```sudo docker exec -it <container-id> /bin/sh

- explore

#### Optimize Container Image Size

There is almost always room for further container image size optimization. Try analysing and reducing the image size using tools like [dive](https://github.com/wagoodman/dive).

#### Lockdown Container Security

Since containers are just processes running on a host machine with some special configuration to provide isolation, careful consideration must be given to security during the application architecture and container definition.

Some good places to start are:

- Docker security: [https://docs.docker.com/engine/security/](https://docs.docker.com/engine/security/)

- Dockerfile best practices: [https://docs.docker.com/develop/develop-images/dockerfile\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile\_best-practices/)

- Docker cheat sheet: [https://github.com/wsargent/docker-cheat-sheet#security](https://github.com/wsargent/docker-cheat-sheet#security)

- Run Docker Bench for Security: [https://github.com/docker/docker-bench-security](https://github.com/docker/docker-bench-security)

- OWASP Docker Security Cheatsheet: [https://cheatsheetseries.owasp.org/cheatsheets/Docker\_Security\_Cheat\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Docker\_Security\_Cheat\_Sheet.html)

### Footnote

Drop me a message through the contact section of this site or via LinkedIn if you have any comments, suggested changes, or see any bugs.