phusion / **passenger-docker**

Docker base images for Ruby, Python, Node.js and Meteor web apps

335 commits | **6** branches | **0** packages | **45** releases | **35** contributors

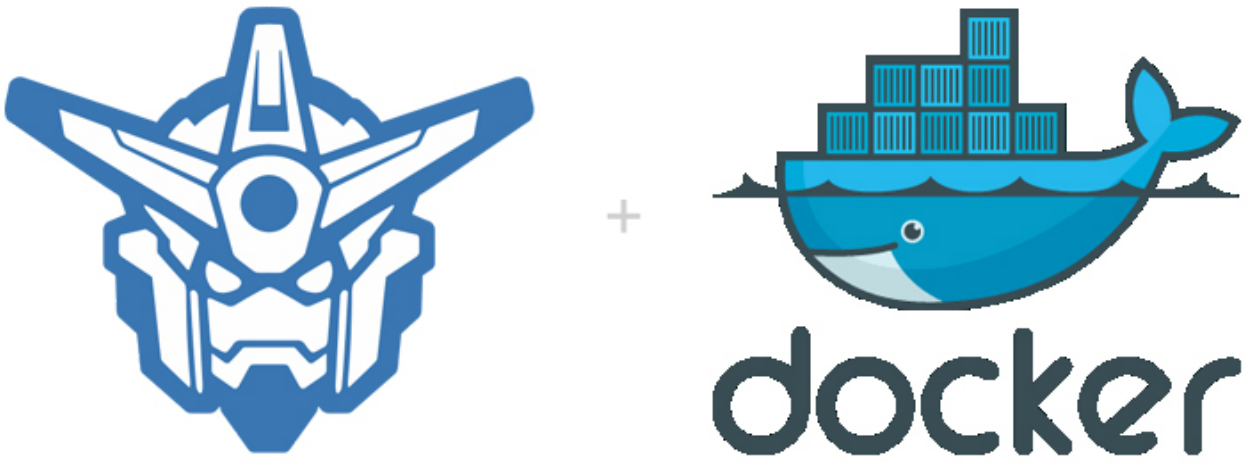
Branch: **master** | **New pull request** | **Create new file** | **Upload files** | **Find file** | **Clone**

CamJN Pre-release tasks — Latest commit `8f6bb02` on

image	Install Phusion Passenger with ruby-2.6.5 (was ruby-2.5.7)	5
test	❤️ Replace #eq with #end_with .	6
.editorconfig	Add editorconfig	
.gitignore	Build against Phusion Passenger 6.0.0	16
.rspec	Add tests	
CHANGELOG.md	Pre-release tasks	4
CODE_OF_CONDUCT.md	Update contact persons	
CONTRIBUTING.md	add Stack Overflow link / tag	
Gemfile	Add tests	
Gemfile.lock	Add tests	
LICENSE.txt	Update copyright statement	
Makefile	Pre-release tasks	4
README.md	📝 Update README.md; default ruby version is 2.6.5	4
Vagrantfile	Always allow Vagrant privisioning, and when provision print the comma...	

README.md

Docker base images for Ruby, Python, Node.js and Meteor web apps



Passenger-docker is a set of Docker images meant to serve as good bases for **Ruby, Python, Node.js and Meteor** w
app images. In line with Phusion Passenger's goal, passenger-docker's goal is to make Docker image building for we
apps much easier and faster.

Why is this image called "passenger"? It's to represent the ease: you just have to sit back and watch most of the hea
lifting being done for you. Passenger-docker is part of a larger and more ambitious project: to make web app deploy
ridiculously simple, to heights never achieved before.

Relevant links: Github | Docker registry | Discussion forum | Twitter | Blog

---

Table of contents

Why use passenger-docker?

Why use passenger-docker instead of doing everything yourself in Dockerfile?

- Your Dockerfile can be smaller.
- It reduces the time needed to write a correct Dockerfile. You won't have to worry about the base system and th stack, you can focus on just your app.
- It sets up the base system **correctly**. It's very easy to get the base system wrong, but this image does everythin correctly. Learn more.
- It drastically reduces the time needed to run `docker build`, allowing you to iterate your Dockerfile more quickl
- It reduces download time during redeploys. Docker only needs to download the base image once: during the firs deploy. On every subsequent deploys, only the changes you make on top of the base image are downloaded.

About the image

What's included?

Passenger-docker is built on top of a solid base: baseimage-docker.

Basics (learn more at baseimage-docker):

- Ubuntu 18.04 LTS as base system.
- A **correct** init process (learn more).
- Fixes APT incompatibilities with Docker.
- syslog-ng.
- The cron daemon.
- Runit for service supervision and management.

Language support:

- Ruby 2.3.8, 2.4.9, 2.5.7 and 2.6.5; JRuby 9.2.0.0.
    - RVM is used to manage Ruby versions. Why RVM?
    - 2.6.5 is configured as the default.
    - JRuby is installed from source, but we register an APT entry for it.
    - JRuby uses OpenJDK 8.
- Python 2.7 and Python 3.6.
- Node.js 10.
- A build system, git, and development headers for many popular libraries, so that the most popular Ruby, Python Node.js native extensions can be compiled without problems.

Web server and application server:

- Nginx 1.14. Disabled by default.
- Phusion Passenger 5. Disabled by default (because it starts along with Nginx).
    - This is a fast and lightweight tool for simplifying web application integration into Nginx.
    - It adds many production-grade features, such as process monitoring, administration and status inspection.
    - It replaces (G)Unicorn, Thin, Puma, uWSGI.
    - Node.js users: watch this 4 minute intro video to learn why it's cool and useful.

Auxiliary services and tools:

- Redis 4.0. Not installed by default.
- Memcached. Not installed by default.

Memory efficiency

Passenger-docker is very lightweight on memory. In its default configuration, it only uses 10 MB of memory (the mem
consumed by bash, runit, syslog-ng, etc).

Image variants

Passenger-docker consists of several images, each one tailor made for a specific user group.

Ruby images

- `phusion/passenger-ruby23` - Ruby 2.3.
- `phusion/passenger-ruby24` - Ruby 2.4.
- `phusion/passenger-ruby25` - Ruby 2.5.
- `phusion/passenger-ruby26` - Ruby 2.6.
- `phusion/passenger-jruby92` - JRuby 9.2.

Node.js and Meteor images

- `phusion/passenger-nodejs` - Node.js 10.

Other images

- `phusion/passenger-full` - Contains everything in the above images. Ruby, Python, Node.js, all in a single ima
your convenience.
- `phusion/passenger-customizable` - Contains only the base system, as described in "What's included?". Ruby
Python and Node.js are not preinstalled. This image is meant to be further customized through your Dockerfile.
example, using this image you can create a custom image that contains only Ruby 2.4, Ruby 2.5 and Node.js.

In the rest of this document we're going to assume that the reader will be using `phusion/passenger-full`, unless
otherwise stated. Simply substitute the name if you wish to use another image.

Inspecting the image

To look around in the image, run:

```
docker run --rm -t -i phusion/passenger-full bash -l
```

You don't have to download anything manually. The above command will automatically pull the passenger-docker im
from the Docker registry.

Using the image as base

Getting started

There are several images, e.g. `phusion/passenger-ruby26` and `phusion/passenger-nodejs`. Choose the one you
See Image variants.

So put the following in your Dockerfile:

```
# Use phusion/passenger-full as base image. To make your builds reproducible, make
```

```
# sure you lock down to a specific version, not to `latest`!
# See https://github.com/phusion/passenger-docker/blob/master/Changelog.md for
# a list of version numbers.
FROM phusion/passenger-full:<VERSION>
# Or, instead of the 'full' variant, use one of these:
#FROM phusion/passenger-ruby23:<VERSION>
#FROM phusion/passenger-ruby24:<VERSION>
#FROM phusion/passenger-ruby25:<VERSION>
#FROM phusion/passenger-ruby26:<VERSION>
#FROM phusion/passenger-jruby92:<VERSION>
#FROM phusion/passenger-nodejs:<VERSION>
#FROM phusion/passenger-customizable:<VERSION>

# Set correct environment variables.
ENV HOME /root

# Use baseimage-docker's init process.
CMD ["/sbin/my_init"]

# If you're using the 'customizable' variant, you need to explicitly opt-in
# for features.
#
# N.B. these images are based on https://github.com/phusion/baseimage-docker,
# so anything it provides is also automatically on board in the images below
# (e.g. older versions of Ruby, Node, Python).
#
# Uncomment the features you want:
#
#   Ruby support
#RUN /pd_build/ruby-2.3.*.sh
#RUN /pd_build/ruby-2.4.*.sh
#RUN /pd_build/ruby-2.5.*.sh
#RUN /pd_build/ruby-2.6.*.sh
#RUN /pd_build/jruby-9.2.*.sh
#   Python support.
#RUN /pd_build/python.sh
#   Node.js and Meteor standalone support.
#   (not needed if you already have the above Ruby support)
#RUN /pd_build/nodejs.sh

# ...put your own build instructions here...

# Clean up APT when done.
RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

The `app` user

The image has an `app` user with UID 9999 and home directory `/home/app`. Your application is supposed to run as
user. Even though Docker itself provides some isolation from the host OS, running applications without root privilege
good security practice.

Your application should be placed inside /home/app.

Note: when copying your application, make sure to set the ownership of the application directory to `app` by calling
`--chown=app:app /local/path/of/your/app /home/app/webapp`

Using Nginx and Passenger

Before using Passenger, you should familiarise yourself with it by reading its documentation.

Nginx and Passenger are disabled by default. Enable them like so:

```
RUN rm -f /etc/service/nginx/down
```

Adding your web app to the image

Passenger works like a `mod_ruby`, `mod_nodejs`, etc. It changes Nginx into an application server and runs your app Nginx. So to get your web app up and running, you just have to add a virtual host entry to Nginx which describes wh you app is, and Passenger will take care of the rest.

You can add a virtual host entry (`server` block) by placing a .conf file in the directory `/etc/nginx/sites-enabled` example:

/etc/nginx/sites-enabled/webapp.conf:

```
server {
    listen 80;
    server_name www.webapp.com;
    root /home/app/webapp/public;

    # The following deploys your Ruby/Python/Node.js/Meteor app on Passenger.

    # Not familiar with Passenger, and used (G)Unicorn/Thin/Puma/pure Node before?
    # Yes, this is all you need to deploy on Passenger! All the reverse proxying,
    # socket setup, process management, etc are all taken care automatically for
    # you! Learn more at https://www.phusionpassenger.com/.
    passenger_enabled on;
    passenger_user app;

    # If this is a Ruby app, specify a Ruby version:
    # For Ruby 2.6
    passenger_ruby /usr/bin/ruby2.6;
    # For Ruby 2.5
    passenger_ruby /usr/bin/ruby2.5;
    # For Ruby 2.4
    passenger_ruby /usr/bin/ruby2.4;
    # For Ruby 2.3
    passenger_ruby /usr/bin/ruby2.3;
}
```

Dockerfile:

```
RUN rm /etc/nginx/sites-enabled/default
ADD webapp.conf /etc/nginx/sites-enabled/webapp.conf
RUN mkdir /home/app/webapp
RUN ...commands to place your web app in /home/app/webapp...
# This copies your web app with the correct ownership.
# COPY --chown=app:app /local/path/of/your/app /home/app/webapp
```

Configuring Nginx

The best way to configure Nginx is by adding .conf files to `/etc/nginx/main.d` and `/etc/nginx/conf.d`. Files in `main.d` are included into the Nginx configuration's main context, while files in `conf.d` are included in the Nginx configuration's http context.

For example:

```
# /etc/nginx/main.d/secret_key.conf:
env SECRET_KEY=123456;
```

```
# /etc/nginx/conf.d/gzip_max.conf:
gzip_comp_level 9;

# Dockerfile:
ADD secret_key.conf /etc/nginx/main.d/secret_key.conf
ADD gzip_max.conf /etc/nginx/conf.d/gzip_max.conf
```

Setting environment variables in Nginx

By default Nginx clears all environment variables (except `TZ`) for its child processes (Passenger being one of them). That's why any environment variables you set with `docker run -e`, Docker linking and `/etc/container_environm` won't reach Nginx.

To preserve these variables, place an Nginx config file ending with `*.conf` in the directory `/etc/nginx/main.d`, in which you tell Nginx to preserve these variables. For example when linking a PostgreSQL container or MongoDB container:

```
# /etc/nginx/main.d/postgres-env.conf:
env POSTGRES_PORT_5432_TCP_ADDR;
env POSTGRES_PORT_5432_TCP_PORT;

# Dockerfile:
ADD postgres-env.conf /etc/nginx/main.d/postgres-env.conf
```

By default, passenger-docker already contains a config file `/etc/nginx/main.d/default.conf` which preserves th `PATH` environment variable.

Application environment name (`RAILS_ENV`, `NODE_ENV`, etc)

Some web frameworks adjust their behavior according to the value some environment variables. For example, Rails respects `RAILS_ENV` while Connect.js respects `NODE_ENV`. By default, Phusion Passenger sets all of the following environment variables to the value **production**:

- `RAILS_ENV`
- `RACK_ENV`
- `WSGI_ENV`
- `NODE_ENV`
- `PASSENGER_APP_ENV`

Setting these environment variables yourself (e.g. using `docker run -e RAILS_ENV=...`) will not have any effect, because Phusion Passenger overrides all of these environment variables. The only exception is `PASSENGER_APP_ENV` below).

With passenger-docker, there are two ways to set the aforementioned environment variables. The first is through th `passenger_app_env` config option in Nginx. For example:

```
# /etc/nginx/sites-enabled/webapp.conf:
server {
    ...
    # Ensures that RAILS_ENV, NODE_ENV, etc are set to "staging"
    # when your application is started.
    passenger_app_env staging;
}
```

The second way is by setting the `PASSENGER_APP_ENV` environment variable from `docker run`

```
docker run -e PASSENGER_APP_ENV=staging YOUR_IMAGE
```

This works because passenger-docker autogenerates an Nginx configuration file (`/etc/nginx/conf.d/00_app_env.conf`) during container boot. This file sets the `passenger_app_env` option in the `http` context. This means that if you already set `passenger_app_env` in the `server` context, running `docker run PASSENGER_APP_ENV=...` won't have any effect!

If you want to set a default value while still allowing that to be overridden by `docker run -e PASSENGER_APP_ENV=`, instead of specifying `passenger_app_env` in your Nginx config file, you should create a `/etc/nginx/conf.d/00_app_env.conf`. This file will be overwritten if the user runs `docker run -e PASSENGER_APP_ENV=...`.

```
# /etc/nginx/conf.d/00_app_env.conf
# File will be overwritten if user runs the container with `-e PASSENGER_APP_ENV=...`!
passenger_app_env staging;
```

Using Redis

Redis is only available in the passenger-customizable and passenger-full images!

Install and enable Redis:

```
# Opt-in for Redis if you're using the 'customizable' image.
#RUN /pd_build/redis.sh

# Enable the Redis service.
RUN rm -f /etc/service/redis/down
```

The configuration file is in /etc/redis/redis.conf. Modify it as you see fit, but make sure `daemonize no` is set.

Using memcached

Memcached is only available in the passenger-customizable and passenger-full images!

Install and enable memcached:

```
# Opt-in for Memcached if you're using the 'customizable' image.
#RUN /pd_build/memcached.sh

# Enable the memcached service.
RUN rm -f /etc/service/memcached/down
```

The configuration file is in /etc/memcached.conf. Note that it does not follow the Debian/Ubuntu format, but our own order to make it work well with runit. The default contents are:

```
# These arguments are passed to the memcached daemon.
MEMCACHED_OPTS="-l 127.0.0.1"
```

Additional daemons

You can add additional daemons to the image by creating runit entries. You only have to write a small shell script wh runs your daemon, and runit will keep it up and running for you, restarting it when it crashes, etc.

The shell script must be called `run`, must be executable, and is to be placed in the directory `/etc/service/<NAME>`

Here's an example showing you how to a memached server runit entry can be made.

```
### In memcached.sh (make sure this file is chmod +x):
#!/bin/sh
# `setuser` is part of baseimage-docker. `setuser mecached xxx...` runs the given command
# (`xxx...`) as the user `memcache`. If you omit this, the command will be run as root.
exec /sbin/setuser memcache /usr/bin/memcached >>/var/log/memcached.log 2>&1

### In Dockerfile:
RUN mkdir /etc/service/memcached
ADD memcached.sh /etc/service/memcached/run
```

Note that the shell script must run the daemon **without letting it daemonize/fork it**. Usually, daemons provide a command line flag or a config file option for that.

Tip: If you're thinking about running your web app, consider deploying it on Passenger instead of on runit. Passenge relieves you from even having to write a shell script, and adds all sorts of useful production features like process sca introspection, etc. These are not available when you're only using runit.

Using Ruby

We use RVM to install and to manage Ruby interpreters. Because of this there are some special considerations you n to know, particularly when you are using the `passenger-full` image which contains multiple Ruby versions installe parallel. You can learn more about RVM at the RVM website, but this section will teach you its basic usage.

Selecting a default Ruby version

The default Ruby (what the `/usr/bin/ruby` command executes) is the latest Ruby version that you've chosen to in You can use RVM select a different version as default.

```
# Ruby 2.3.8
RUN bash -lc 'rvm --default use ruby-2.3.8'
# Ruby 2.4.9
RUN bash -lc 'rvm --default use ruby-2.4.9'
# Ruby 2.5.7
RUN bash -lc 'rvm --default use ruby-2.5.7'
# Ruby 2.6.5
RUN bash -lc 'rvm --default use ruby-2.6.5'
# JRuby 9.2.0.0
RUN bash -lc 'rvm --default use jruby-9.2.0.0'
```

Learn more: RVM: Setting the default Ruby.

Running a command with a specific Ruby version

You can run any command with a specific Ruby version by prefixing it with `rvm-exec <IDENTIFIER>`. For example:

```
$ rvm-exec 2.5.7 ruby -v
ruby 2.5.7
$ rvm-exec 2.4.9 ruby -v
ruby 2.4.9
```

More examples, but with Bundler instead:

```
# This runs 'bundle install' using Ruby 2.5.7
rvm-exec 2.5.7 bundle install

# This runs 'bundle install' using Ruby 2.4.9
rvm-exec 2.4.9 bundle install
```

Default wrapper scripts

Rubies are installed by RVM to /usr/local/rvm. Interactive and login Bash shells load the RVM environment, which ens
that the appropriate directories under /usr/local/rvm are in PATH.

But this means that if you invoke a command without going through an interactive and login Bash shell (e.g. directly
`docker exec`) then the RVM environment won't be loaded. In order to make Ruby work even in this case, Passenge
docker includes a bunch of wrapper scripts:

- /usr/bin/ruby
- /usr/bin/rake
- /usr/bin/gem
- /usr/bin/bundle

These wrapper scripts execute the respective command through `rvm-exec` using the default Ruby interpreter.

Running scripts during container startup

passenger-docker uses the baseimage-docker init system, `/sbin/my_init`. This init system runs the following scri
during startup, in the following order:

- All executable scripts in `/etc/my_init.d`, if this directory exists. The scripts are run during in lexicographic or
- The script `/etc/rc.local`, if this file exists.

All scripts must exit correctly, e.g. with exit code 0. If any script exits with a non-zero exit code, the booting will fail.

The following example shows how you can add a startup script. This script simply logs the time of boot to the file
/tmp/boottime.txt.

```
### In logtime.sh (make sure this file is chmod +x):
#!/bin/sh
date > /tmp/boottime.txt

### In Dockerfile:
RUN mkdir -p /etc/my_init.d
ADD logtime.sh /etc/my_init.d/logtime.sh
```

Upgrading the operating system inside the container

passenger-docker images contain an Ubuntu 16.04 operating system. You may want to update this OS from time to
for example to pull in the latest security updates. OpenSSL is a notorious example. Vulnerabilities are discovered in
OpenSSL on a regular basis, so you should keep OpenSSL up-to-date as much as you can.

While we release passenger-docker images with the latest OS updates from time to time, you do not have to rely on
You can update the OS inside passenger-docker images yourself, and it is recommend that you do this instead of wa
for us.

To upgrade the OS in the image, run this in your Dockerfile:

```
RUN apt-get update && apt-get upgrade -y -o Dpkg::Options::="--force-confold"
```

Upgrading Passenger to the latest version

Upgrading to the latest image

Passenger-docker images contain a specific Passenger version by default. We regularly update passenger-docker w
the latest version of Passenger, Ruby, Node.js, etc.

To upgrade your image to the latest passenger-docker version, please edit your Dockerfile and change the passeng
docker version in the `FROM` command to the latest version. You can find a list of available versions in the Changelog

For example, if you were using passenger-docker 0.9.16 and want to upgrade to 0.9.17, then change...

```
FROM phusion/passenger-docker-XXXX:0.9.16
```

...to:

```
FROM phusion/passenger-docker-XXXX:0.9.17
```

Then rebuild your image.

Upgrading Passenger without waiting for image updates

We do not update the passenger-docker image on *every* Passenger release. Having said that, you can upgrade Pass
at any time, without waiting for us to release a new image.

Passenger is installed through the Passenger APT repository, so you can use APT to upgrade Passenger.

To upgrade to the latest Passenger version, run this to your Dockerfile:

```
RUN apt-get update && apt-get upgrade -y -o Dpkg::Options::="--force-confold"
```

Container administration

One of the ideas behind Docker is that containers should be stateless, easily restartable, and behave like a black bo
However, you may occasionally encounter situations where you want to login to a container, or to run a command ins
container, for development, inspection and debugging purposes. This section describes how you can administer the
container for those purposes.

***Tip**: passenger-docker is based on baseimage-docker. Please consult the baseimage-docker documentation for mo
container administration documentation and tips.*

Running a one-shot command in a new container

***Note:** This section describes how to run a command insider a -new- container. To run a command inside an existing
running container, see Running a command in an existing, running container.*

Normally, when you want to create a new container in order to run a single command inside it, and immediately exit a
the command exits, you invoke Docker like this:

```
docker run YOUR_IMAGE COMMAND ARGUMENTS...
```

However the downside of this approach is that the init system is not started. That is, while invoking `COMMAND`, impor
daemons such as cron and syslog are not running. Also, orphaned child processes are not properly reaped, because
`COMMAND` is PID 1.

Passenger-docker provides a facility to run a single one-shot command, while solving all of the aforementioned prob
Run a single command in the following manner:

```
docker run YOUR_IMAGE /sbin/my_init -- COMMAND ARGUMENTS ...
```

This will perform the following:

- Runs all system startup files, such as /etc/my_init.d/* and /etc/rc.local.
- Starts all runit services.
- Runs the specified command.
- When the specified command exits, stops all runit services.

For example:

```
$ docker run phusion/passenger-full:<VERSION> /sbin/my_init -- ls
*** Running /etc/rc.local...
*** Booting runit daemon...
*** Runit started as PID 80
*** Running ls...
bin  boot  dev  etc  home  image  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  sr
sys  tmp  usr  var
*** ls exited with exit code 0.
*** Shutting down runit daemon (PID 80)...
*** Killing all processes...
```

You may find that the default invocation is too noisy. Or perhaps you don't want to run the startup files. You can cust all this by passing arguments to `my_init`. Invoke `docker run YOUR_IMAGE /sbin/my_init --help` for more information.

The following example runs `ls` without running the startup files and with less messages, while running all runit serv

```
$ docker run phusion/passenger-full:<VERSION> /sbin/my_init --skip-startup-files --quiet -- ls
bin  boot  dev  etc  home  image  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  sr
sys  tmp  usr  var
```

Running a command in an existing, running container

There are two ways to run a command inside an existing, running container.

- Through the `docker exec` tool. This is builtin Docker tool, available since Docker 1.4. Internally, it uses Linux ke system calls in order to execute a command within the context of a container. Learn more in Login to the contai running a command inside it, via `docker exec`.
- Through SSH. This approach requires running an SSH daemon inside the container, and requires you to setup S keys. Learn more in Login to the container, or running a command inside it, via SSH.

Both way have their own pros and cons, which you can learn in their respective subsections.

Login to the container, or running a command inside it, via `docker exec`

You can use the `docker exec` tool on the Docker host OS to login to any container that is based on passenger-doc You can also use it to run a command inside a running container. `docker exec` works by using Linux kernel system

Here's how it compares to using SSH to login to the container or to run a command inside it:

- Pros
  - Does not require running an SSH daemon inside the container.
  - Does not require setting up SSH keys.
  - Works on any container, even containers not based on passenger-docker.
- Cons
  - If the `docker exec` process on the host is terminated by a signal (e.g. with the `kill` command or even w Ctrl-C), then the command that is executed by `docker exec` is *not* killed and cleaned up. You will either ha do that manually, or you have to run `docker exec` with `-t -i`.
  - Requires privileges on the Docker host to be able to access the Docker daemon. Note that anybody who ca

access the Docker daemon effectively has root access.
    - Not possible to allow users to login to the container without also letting them login to the Docker host.

Usage

Start a container:

```
docker run YOUR_IMAGE
```

Find out the ID of the container that you just ran:

```
docker ps
```

Now that you have the ID, you can use `docker exec` to run arbitrary commands in the container. For example, to ru
`echo hello world`:

```
docker exec YOUR-CONTAINER-ID echo hello world
```

To open a bash session inside the container, you must pass `-t -i` so that a terminal is available:

```
docker exec -t -i YOUR-CONTAINER-ID bash -l
```

Login to the container, or running a command inside it, via SSH

You can use SSH to login to any container that is based on passenger-docker. You can also use it to run a command
inside a running container.

Here's how it compares to using `docker exec` to login to the container or to run a command inside it:

- Pros
    - Does not require root privileges on the Docker host.
    - Allows you to let users login to the container, without letting them login to the Docker host. However, this is
      enabled by default because passenger-docker does not expose the SSH server to the public Internet by de
- Cons
    - Requires setting up SSH keys. However, passenger-docker makes this easy for many cases through a
      pregenerated, insecure key. Read on to learn more.

Enabling SSH

Passenger-docker disables the SSH server by default. Add the following to your Dockerfile to enable it:

```
RUN rm -f /etc/service/sshd/down

# Regenerate SSH host keys. Passenger-docker does not contain any, so you
# have to do that yourself. You may also comment out this instruction; the
# init system will auto-generate one during boot.
RUN /etc/my_init.d/00_regen_ssh_host_keys.sh
```

About SSH keys

First, you must ensure that you have the right SSH keys installed inside the container. By default, no keys are installe
nobody can login. For convenience reasons, we provide a pregenerated, insecure key (PuTTY format) that you can e
enable. However, please be aware that using this key is for convenience only. It does not provide any security becau
this key (both the public and the private side) is publicly available. **In production environments, you should use you
own keys**.

Using the insecure key for one container only

You can temporarily enable the insecure key for one container only. This means that the insecure key is installed at
container boot. If you `docker stop` and `docker start` the container, the insecure key will still be there, but if you
`docker run` to start a new container then that container will not contain the insecure key.

Start a container with `--enable-insecure-key`:

```
docker run YOUR_IMAGE /sbin/my_init --enable-insecure-key
```

Find out the ID of the container that you just ran:

```
docker ps
```

Once you have the ID, look for its IP address with:

```
docker inspect -f "{{ .NetworkSettings.IPAddress }}" <ID>
```

Now that you have the IP address, you can use SSH to login to the container, or to execute a command inside it:

```
# Download the insecure private key
curl -o insecure_key -fSL https://raw.githubusercontent.com/phusion/baseimage-docker/master/image/
chmod 600 insecure_key

# Login to the container
ssh -i insecure_key root@<IP address>

# Running a command inside the container
ssh -i insecure_key root@<IP address> echo hello world
```

Enabling the insecure key permanently

It is also possible to enable the insecure key in the image permanently. This is not generally recommended, but is su
for e.g. temporary development or demo environments where security does not matter.

Edit your Dockerfile to install the insecure key permanently:

```
RUN /usr/sbin/enable_insecure_key
```

Instructions for logging in the container is the same as in section Using the insecure key for one container only.

Using your own key

Edit your Dockerfile to install an SSH public key:

```
## Install an SSH of your choice.
ADD your_key.pub /tmp/your_key.pub
RUN cat /tmp/your_key.pub >> /root/.ssh/authorized_keys && rm -f /tmp/your_key.pub
```

Then rebuild your image. Once you have that, start a container based on that image:

```
docker run your-image-name
```

Find out the ID of the container that you just ran:

```
docker ps
```

Once you have the ID, look for its IP address with:

```
docker inspect -f "{{ .NetworkSettings.IPAddress }}" <ID>
```

Now that you have the IP address, you can use SSH to login to the container, or to execute a command inside it:

```
# Login to the container
ssh -i /path-to/your_key root@<IP address>

# Running a command inside the container
ssh -i /path-to/your_key root@<IP address> echo hello world
```

The `docker-ssh` tool

Looking up the IP of a container and running an SSH command quickly becomes tedious. Luckily, we provide the `docker-ssh` tool which automates this process. This tool is to be run on the *Docker host*, not inside a Docker container.

First, install the tool on the Docker host:

```
curl --fail -L -O https://github.com/phusion/baseimage-docker/archive/master.tar.gz && \
tar xzf master.tar.gz && \
sudo ./baseimage-docker-master/install-tools.sh
```

Then run the tool as follows to login to a container using SSH:

```
docker-ssh YOUR-CONTAINER-ID
```

You can lookup `YOUR-CONTAINER-ID` by running `docker ps`.

By default, `docker-ssh` will open a Bash session. You can also tell it to run a command, and then exit:

```
docker-ssh YOUR-CONTAINER-ID echo hello world
```

Inspecting the status of your web app

If you use Passenger to deploy your web app, run:

```
passenger-status
passenger-memory-stats
```

Logs

If anything goes wrong, consult the log files in /var/log. The following log files are especially important:

- /var/log/nginx/error.log

- /var/log/syslog
- Your app's log file in /home/app.

Switching to Phusion Passenger Enterprise

If you are a Phusion Passenger Enterprise customer, then you can switch to the Enterprise variant as follows.

1. Login to the Customer Area.
2. Download the license key and store it in the same directory as your Dockerfile.
3. Insert into your Dockerfile:

```
ADD passenger-enterprise-license /etc/passenger-enterprise-license
RUN echo deb https://download:$DOWNLOAD_TOKEN@www.phusionpassenger.com/enterprise_apt bionic main
RUN apt-get update && apt-get install -y -o Dpkg::Options::="--force-confold" passenger-enterpris
```

```
Replace `$DOWNLOAD_TOKEN` with your actual download token, as found in the Customer Area.
```

Building the image yourself

If for whatever reason you want to build the image yourself instead of downloading it from the Docker registry, follow these instructions.

Clone this repository:

```
git clone https://github.com/phusion/passenger-docker.git
cd passenger-docker
```

Start a virtual machine with Docker in it. You can use the Vagrantfile that we've already provided.

```
vagrant up
vagrant ssh
cd /vagrant
```

Build one of the images:

```
make build_ruby23
make build_ruby24
make build_ruby25
make build_ruby26
make build_jruby92
make build_nodejs
make build_customizable
make build_full
```

If you want to call the resulting image something else, pass the NAME variable, like this:

```
make build NAME=joe/passenger
```

FAQ

Why are you using RVM? Why not rbenv or chruby?

In summary:

- We have found RVM to be much more user friendly than rbenv and chruby.
- RVM supplies precompiled binaries, while rbenv and chruby only support compiling Ruby from source.
- Installing Ruby from Brightbox's APT repository caused too many problems. We used Brightbox's APT repositor the past, but we concluded that it is not the way to go forward.

Rbenv and chruby's main value proposition is that they are "simple". Indeed, they are simpler in implementation (few lines of code) than RVM, but they are not simpler to use. Rbenv and chruby are built on the Unix "do one thing only" philosophy. While this is sound, it is not necessarily the behavior that users want: I have seen many users struggling basic rbenv/chruby usage because of lack of understanding of environment variables, or not having installed the rig dependencies. Many users do not understand how the system is supposed to function and what all the different par so doing one thing only may not be what they need. In such a case the simplicity ends up being more of a liability th asset. It's like selling a car engine, frame and interior separately, while most consumers want an entire car.

RVM is built around a more "holistic" philosophy, if you will. It tries harder to be friendly to users who may not neces understand how everything works, for example by automatically installing a bash profile entry, by automatically insta necessary dependencies.

Another critique of RVM is that it is complicated and causes problems. This has not been our experience: perhaps th was the case in the past, but we have found RVM to be quite stable.

Why don't you just install Ruby manually from source?

By installing Ruby manually from source, we are just reinventing some of the functionality provided by a real Ruby ve manager such as RVM, so we may as well use one to save ourselves time. There is no reason not to use RVM: it only occupies 5 MB of space.

Why are you not using the Brightbox's APT repository?

The Brightbox APT repository contains packages for multiple Ruby versions, which can be installed side-by-side. At this seems like the perfect solution. And indeed, passenger-docker used to use the Brightbox APT repository.

Unfortunately, we have found that it is much harder to make the different Rubies play nice with each other than it sh be. Despite being installable side-to-side, they still conflict with each other. The most notable problem is that all Rub RubyGems install binwrappers to /usr/local/bin, but binwrappers generated by different Ruby versions may not be compatible with each other.

RVM provides much better isolation between different Ruby versions.

Why don't you just install Ruby from Ubuntu's APT repository?

Because we need to support Ruby versions not available from Ubuntu's APT repository. Besides, Ubuntu (and Debia notorious for being slow with updating Ruby packages. By the time the next Ruby version is released, we will have to until the next Ubuntu LTS version before we can use it.

Contributing

Thanks for your interest in contributing! There are many ways to contribute to this project. Get started here.

Conclusion

- Using passenger-docker? Tweet about us or follow us on Twitter.
- Having problems? Please post a message at the discussion forum.
- Looking for a minimal image containing only a correct base system? Take a look at baseimage-docker.
- Need a helping hand? Phusion also offers consulting on a wide range of topics, including Web Development, UI/ Research & Design, Technology Migration and Auditing.

Please enjoy passenger-docker, a product by Phusion. :-)