Phusion Passenger users guide

Table of Contents

Phusion Passenger is an Apache module, which makes deploying Ruby and Ruby on Rails applications on Apache a breeze. It follows the usual Ruby on Rails conventions, such as "Don't-Repeat-Yourself" and ease of setup, while at the same time providing enough flexibility.

This users guide will teach you:

- How to install Phusion Passenger.
- How to configure Phusion Passenger.
- How to deploy a Ruby on Rails application.
- How to deploy a Rack-based Ruby application.
- How to solve common problems.

This guide assumes that the reader is somewhat familiar with Apache and with using the commandline.

1. Support information

1.1. Supported operating systems

Phusion Passenger works on any POSIX-compliant operating system. In other words: practically any operating system on earth, except Microsoft Windows.

Phusion Passenger is confirmed on a large number of operating systems and Linux distributions, including, but not limited to, Ubuntu, Debian, CentOS/Fedora/RHEL, Gentoo,

Mac OS X, FreeBSD and Solaris. Both 32-bit and 64-bit platforms are supported.

The only POSIX-compliant operating system on which Phusion Passenger for Apache is known not to work at this time, is OpenBSD. Please use Phusion Passenger for Nginx instead.

Please report a bug or join our discussion forum if it doesn't work on your POSIX-compliant operating system.

1.2. Where to get support

- Issue tracker - report bugs here.
- Discussion forum - post a message here if you're experiencing problems.

2. Installing, upgrading and uninstalling Phusion Passenger

2.1. Generic installation instructions

2.1.1. Overview of installation methods

There are three ways to install Phusion Passenger:

1. By installing the Phusion Passenger gem, as instructed on the "Install" page on the Phusion Passenger website.
2. By downloading the source tarball from the Phusion Passenger website (*passenger-x.x.x.tar.gz*).
3. By installing a native Linux package (e.g. Debian package).

The following sections will explain each installation method. Please read the section for the installation method that you prefer. In our opinion, installing the gem or the native package is easiest. For these two installation methods, Phusion Passenger provides an easy-to-use installer.

2.1.2. Preparation (gem and source tarball only)

If you want to install Phusion Passenger via the gem or the source tarball, then some preparations might be required. You can skip this subsection if you're installing Phusion Passenger via a native Linux package, because no compilation is necessary.

Switching to a root command prompt

Before installing, you will probably need to switch to the `root` user first. When you install Phusion Passenger via a gem or a source tarball, some Phusion Passenger files have to be compiled, which requires write access to the directory in which the Phusion Passenger files are located. On Unix systems, the root user is the user who has write access to the entire system. So unless you know that your normal user account has write access to the Phusion Passenger directory, you should switch to root before installing Phusion Passenger.

You can switch to root by typing the following command:

```
sudo -s
```

This will open a command prompt as the root user, from which you can proceed with installing Phusion Passenger.

If your system does not have *sudo* installed, please type the following command instead, which should do the same thing:

```
su
```

Specifying the correct Apache installation

The Phusion Passenger installer will attempt to automatically detect Apache, and compile Phusion Passenger against that Apache version. It does this by looking for the `apxs` or `apxs2` command in the PATH environment variable. Apxs is an integral part of any Apache installation.

However, some systems have multiple Apache installations. This is likely the case on MacOS X: the OS ships with Apache, but users tend to install another Apache version seperately, e.g. via MacPorts. If your system has multiple Apache installations, then you will need to tell the Phusion Passenger installer which one to use. It is very important that you specify the correct Apache installation, because if you load Phusion Passenger in an Apache installation that it wasn't compiled against, then it will likely crash.

On yet other systems, Apache is installed in a non-standard location, preventing the Phusion Passenger installer from detecting Apache. This is most likely the case on systems on which Apache was installed by hand from source, i.e. as opposed to installed through the system's native package manager. If this is the case, then you will also have to tell the installer where it can find Apache.

To do so, set the `APXS2` environment variable to the full path of the correct `apxs` or `apxs2` command. Suppose that you want to use the Apache installation in */opt/apache2*. Then, assuming that the corresponding `apxs` program's path is */opt/apache2/bin/apxs*, type:

```
export APXS2=/opt/apache2/bin/apxs
```

On some systems, the `apxs` program might be called `apxs2`, and it might be located in the `sbin` folder instead of the `bin` folder.

Environment variables and *sudo*

By default, the *sudo* command will erase any environment variables that it doesn't recognize, prior to executing the given command. So if you set APXS2 as a normal user, then run `sudo passenger-install-apache2-module` (which is the command for the Phusion Passenger installer), then the installer will not receive the environment variable value that you set. To solve this problem, please become root prior to setting any environment variables, as described in the previous subsection.

Specifying the correct Ruby installation

If your system has multiple Ruby installations — which is likely the case on MacOS X, or if you've also installed Ruby Enterprise Edition — then you will need to tell the operating system which Ruby installation to use, prior to running the Phusion Passenger installer. If you only have one Ruby installation (the case on most Linux systems), then you can skip this section because Phusion Passenger will automatically detect it.

To specify a Ruby installation, prepend your Ruby installation's `bin` directory to the `PATH` environment variable. For example, if you have the following Ruby installations:

- /usr/bin/ruby
- /opt/myruby/bin/ruby

and you want to use the latter, then type:

```
export PATH=/opt/myruby/bin:$PATH
```

2.1.3. Installing via the gem

Please install the gem and then run the Phusion Passenger installer, by typing the following commands:

```
gem install passenger-x.x.x.gem
passenger-install-apache2-module
```

Please follow the instructions given by the installer.

2.1.4. Installing via the source tarball

Extract the tarball to whatever location you prefer. **The Phusion Passenger files are to reside in that location permanently.** For example, if you would like Phusion Passenger to reside in `/opt/passenger-x.x.x`:

```
cd /opt
tar xzvf ~/YourDownloadsFolder/passenger-x.x.x.tar.gz
```

Next, run the included installer:

```
/opt/passenger-x.x.x/bin/passenger-install-apache2-module
```

Please follow the instructions given by the installer.

Please do not remove the *passenger-x.x.x* folder after installation.
Furthermore, the *passenger-x.x.x* folder must be accessible by Apache.

2.1.5. Installing via a native Linux package

John Leach from Brightbox has kindly provided an Ubuntu Hardy package for Phusion Passenger. The package is available from the Brightbox repository.

Please install the native Linux package, e.g.:

```
sudo sh -c 'echo "deb http://apt.brightbox.net hardy main" > /etc/apt/sources.li
sudo sh -c 'wget -q -O - http://apt.brightbox.net/release.asc | apt-key add -'
sudo apt-get update
sudo apt-get install libapache2-mod-passenger
```

2.1.6. What does the installer do?

Although we call it an “installer”, it doesn’t actually install anything. The installer checks whether all required dependencies are installed, compiles Phusion Passenger for you, and tells you how to modify the Apache configuration file, but it doesn’t copy any files around.

`passenger-install-apache2-module` is actually just a user-friendly frontend around the command `rake apache2`, which performs the actual compilation of Phusion Passenger.

2.2. Operating system-specific instructions and information

2.2.1. MacOS X

Ben Ruebenstein has written an excellent tutorial on installing Phusion Passenger on OS X.

2.2.2. Ubuntu Linux

Ben Hughes has written an article on installing Phusion Passenger on Ubuntu.

2.2.3. OpenSolaris

J Aaron Farr has written a guide about setting up Ruby on Rails and Phusion Passenger on OpenSolaris and EC2.

2.3. Upgrading or downgrading Phusion Passenger

2.3.1. Via a gem or a source tarball

To ugrade or downgrade Phusion Passenger via the gem or the source tarball, install the newer or older version as you normally would; that is, install the gem or unpack the tarball, and run `passenger-install-apache2-module`. Eventually `passenger-install-apache2-module` will tell you to copy & paste some settings into the Apache configuration file; something that looks along the lines of:

```
LoadModule passenger_module ...
PassengerRoot ...
PassengerRuby ...
```

Because you already had Phusion Passenger installed, you already had similar settings in your Apache configuration file, just with different values. **Replace** the old settings with the new ones that the installer outputs. It is important that the old settings are removed, otherwise Phusion Passenger may malfunction.

When you're done, restart Apache.

2.3.2. Via a native Linux package

There are no special instructions required to upgrade or downgrade Phusion Passenger via a native Linux package.

2.4. Unloading (disabling) Phusion Passenger from Apache without uninstalling it

You can temporarily unload (disable) Phusion Passenger from Apache, without uninstalling the Phusion Passenger files, so that Apache behaves as if Phusion Passenger was never installed in the first place. This might be useful to you if, for example, you seem to be experiencing a problem caused by Phusion Passenger, but you want to make sure whether that's actually the case, without having to through the hassle of uninstalling Phusion Passenger completely. When disabled, Phusion Passenger will not occupy any memory or CPU or otherwise interfere with Apache.

To unload Phusion Passenger from Apache, edit your Apache configuration file(s) and comment out:

- all Phusion Passenger configuration directives.
- the *LoadModule passenger_module* directive.

For example, if your configuration file looks like this...

```
Listen *:80
NameVirtualHosts *:80
....
```

```
LoadModule passenger_module /somewhere/passenger-x.x.x/ext/apache2/mod_passenger

PassengerRuby /usr/bin/ruby
PassengerRoot /somewhere/passenger/x.x.x
PassengerMaxPoolSize 10

<VirtualHost *:80>
    ServerName www.foo.com
    DocumentRoot /webapps/foo/public
    RailsBaseURI /rails
</VirtualHost>
```

...then comment out the relevant directives, so that it looks like this:

```
Listen *:80
NameVirtualHosts *:80
....

# LoadModule passenger_module /somewhere/passenger-x.x.x/ext/apache2/mod_passeng

# PassengerRuby /usr/bin/ruby
# PassengerRoot /somewhere/passenger/x.x.x
# PassengerMaxPoolSize 10

<VirtualHost *:80>
    ServerName www.foo.com
    DocumentRoot /webapps/foo/public
    # RailsBaseURI /rails
</VirtualHost>
```

After you've done this, save the file and restart Apache.

2.5. Uninstalling Phusion Passenger

To uninstall Phusion Passenger, please first remove all Phusion Passenger configuration directives from your Apache configuration file(s). After you've done this, you need to remove the Phusion Passenger files.

- If you installed Phusion Passenger via a gem, then type `gem uninstall passenger`. You might have to run this as root.
- If you installed Phusion Passenger via a source tarball, then remove the directory in which you placed the extracted Phusion Passenger files. This directory is the same as the one pointed to the by *PassengerRoot* configuration directive.
- If you installed Phusion Passenger via a Debian package, then remove type `sudo apt-get remove libapache2-mod-passenger`.

3. Deploying a Ruby on Rails application

Suppose you have a Ruby on Rails application in */webapps/mycook*, and you own the domain *www.mycook.com*. You can either deploy your application to the virtual host's root (i.e. the application will be accessible from the root URL, *http://www.mycook.com/*), or in a sub URI (i.e. the application will be accessible from a sub URL, such as *http://www.mycook.com/railsapplication*).

The default `RAILS_ENV` environment in which deployed Rails applications are run, is "production". You can change this by changing the *RailsEnv* configuration option.

3.1. Deploying to a virtual host's root

Add a virtual host entry to your Apache configuration file. Make sure that the following conditions are met:

- The virtual host's document root must point to your Ruby on Rails application's *public* folder.
- The Apache per-directory permissions must allow access to this folder.
- MultiViews must be disabled for this folder.

For example:

```
<VirtualHost *:80>
    ServerName www.mycook.com
    DocumentRoot /webapps/mycook/public
    <Directory /webapps/mycook/public>
        Allow from all
        Options -MultiViews
    </Directory>
</VirtualHost>
```

You may also need to tweak your file/folder permissions. Make sure that the following folders are readable and executable by Apache:

- this *public* folder.
- the application's *config* folder.
- all parent folders. That is, /webapps/mycook and /webapps must also be readable and executable by Apache.

Then restart Apache. The application has now been deployed.

3.2. Deploying to a sub URI

Suppose that you already have a virtual host:

```
<VirtualHost *:80>
    ServerName www.phusion.nl
    DocumentRoot /websites/phusion
    <Directory /websites/phusion>
        Allow from all
    </Directory>
</VirtualHost>
```

And you want your Ruby on Rails application to be accessible from the URL *http://www.phusion.nl/rails*.

To do this, make a symlink in the virtual host's document root, and have it point to your Ruby on Rails application's *public* folder. For example:

```
ln -s /webapps/mycook/public /websites/phusion/rails
```

Next, add a RailsBaseURI option to the virtual host configuration, and also make sure that:

- The Apache per-directory permissions allow access to this folder.
- MultiViews is disabled for this folder.

For example:

```
<VirtualHost *:80>
    ServerName www.phusion.nl
    DocumentRoot /websites/phusion
    <Directory /websites/phusion>
        Allow from all
    </Directory>

    RailsBaseURI /rails                    # <-- These lines have
    <Directory /websites/phusion/rails>    # <-- been added.
        Options -MultiViews                # <--
    </Directory>                           # <--
</VirtualHost>
```

Then restart Apache. The application has now been deployed.

If you're deploying to a sub-URI then please make sure that your view templates correctly handles references to sub-URI static assets! Otherwise you may find broken links to images, CSS files, JavaScripts, etc. Please read How to fix broken images/CSS/JavaScript URIs in sub-URI deployments for more information.

You can deploy multiple Rails applications under a virtual host, by specifying RailsBaseURI multiple times. For example:

```
<VirtualHost *:80>
    ....
    RailsBaseURI /app1
    RailsBaseURI /app2
    RailsBaseURI /app3
</VirtualHost>
```

3.3. Redeploying (restarting the Ruby on Rails application)

Deploying a new version of a Ruby on Rails application is as simple as re-uploading the application files, and restarting the application.

There are two ways to restart the application:

1. By restarting Apache.
2. By creating or modifying the file *tmp/restart.txt* in the Rails application's root folder. Phusion Passenger will automatically restart the application during the next request.

For example, to restart our example MyCook application, we type this in the command line:

```
touch /webapps/mycook/tmp/restart.txt
```

Please note that, unlike earlier versions of Phusion Passenger, *restart.txt* is not automatically deleted. Phusion Passenger checks whether the timestamp of this file has changed in order to determine whether the application should be restarted.

3.4. Migrations

Phusion Passenger is not related to Ruby on Rails migrations in any way. To run migrations on your deployment server, please login to your deployment server (e.g. with *ssh*) and type `rake db:migrate RAILS_ENV=production` in a shell console, just like one would normally run migrations.

3.5. Capistrano integration

See Capistrano recipe.

4. Deploying a Rack-based Ruby application

Phusion Passenger supports arbitrary Ruby web applications that follow the Rack interface.

Phusion Passenger assumes that Rack application directories have a certain layout. Suppose that you have a Rack application in */webapps/rackapp*. Then that folder must contain at least three entries:

- *config.ru*, a Rackup file for starting the Rack application. This file must contain the complete logic for initializing the application.
- *public/*, a folder containing public static web assets, like images and stylesheets.
- *tmp/*, used for *restart.txt* (our application restart mechanism). This will be explained in a following subsection.

So */webapps/rackapp* must, at minimum, look like this:

```
/webapps/rackapp
  |
  +-- config.ru
  |
  +-- public/
  |
  +-- tmp/
```

Suppose you own the domain *www.rackapp.com*. You can either deploy your application to the virtual host's root (i.e. the application will be accessible from the root URL, *http://www.rackapp.com/*), or in a sub URI (i.e. the application will be accessible from a sub URL, such as *http://www.rackapp.com/rackapp*).

The default `RACK_ENV` environment in which deployed Rack applications are run, is "production". You can change this by changing the *RackEnv* configuration option.

4.1. Tutorial/example: writing and deploying a Hello World Rack

application

First we create a Phusion Passenger-compliant Rack directory structure:

```
$ mkdir /webapps/rack_example
$ mkdir /webapps/rack_example/public
$ mkdir /webapps/rack_example/tmp
```

Next, we write a minimal "hello world" Rack application:

```
$ cd /webapps/rack_example
$ some_awesome_editor config.ru
...type in some source code...
$ cat config.ru
app = proc do |env|
    [200, { "Content-Type" => "text/html" }, ["hello <b>world</b>"]]
end
run app
```

Finally, we deploy it by adding the following configuration options to the Apache configuration file:

```
<VirtualHost *:80>
    ServerName www.rackexample.com
    DocumentRoot /webapps/rack_example/public
    <Directory /webapps/rack_example/public>
        Allow from all
        Options -MultiViews
    </Directory>
</VirtualHost>
```

And we're done! After an Apache restart, the above Rack application will be available under the URL *http://www.rackexample.com/*.

4.2. Deploying to a virtual host's root

Add a virtual host entry to your Apache configuration file. Make sure that the following conditions are met:

- The virtual host's document root must point to your Rack application's *public* folder.
- The Apache per-directory permissions must allow access to this folder.
- MultiViews must be disabled for this folder.

For example:

```
<VirtualHost *:80>
    ServerName www.rackapp.com
    DocumentRoot /webapps/rackapp/public
    <Directory /webapps/rackapp/public>
        Allow from all
        Options -MultiViews
    </Directory>
</VirtualHost>
```

You may also need to tweak your file/folder permissions. Make sure that the following folders are readable and executable by Apache:

- this *public* folder.
- the application's *config* folder.
- all parent folders. That is, /webapps/rackapp and /webapps must also be readable and executable by Apache.

Then restart Apache. The application has now been deployed.

4.3. Deploying to a sub URI

Suppose that you already have a virtual host:

```
<VirtualHost *:80>
    ServerName www.phusion.nl
    DocumentRoot /websites/phusion
    <Directory /websites/phusion>
        Allow from all
    </Directory>
</VirtualHost>
```

And you want your Rack application to be accessible from the URL *http://www.phusion.nl/rack*.

To do this, make a symlink in the virtual host's document root, and have it point to your Rack application's *public* folder. For example:

```
ln -s /webapps/rackapp/public /websites/phusion/rack
```

Next, add a RackBaseURI option to the virtual host configuration, and also make sure that:

- The Apache per-directory permissions allow access to this folder.
- MultiViews is disabled for this folder.

For example:

```
<VirtualHost *:80>
    ServerName www.phusion.nl
    DocumentRoot /websites/phusion
    <Directory /websites/phusion>
        Allow from all
    </Directory>

    RackBaseURI /rails                    # <-- These lines have
    <Directory /websites/phusion/rails>   # <-- been added.
        Options -MultiViews               # <--
    </Directory>                          # <--
</VirtualHost>
```

Then restart Apache. The application has now been deployed.

You can deploy multiple Rack applications under a virtual host, by specifying RackBaseURI multiple times. For example:

```
<VirtualHost *:80>
    ....
    RackBaseURI /app1
    RackBaseURI /app2
    RackBaseURI /app3
</VirtualHost>
```

4.4. Redeploying (restarting the Rack application)

Deploying a new version of a Rack application is as simple as re-uploading the application files, and restarting the application.

There are two ways to restart the application:

1. By restarting Apache.
2. By creating or modifying the file *tmp/restart.txt* in the Rack application's root folder. Phusion Passenger will automatically restart the application.

For example, to restart our example application, we type this in the command line:

```
touch /webapps/rackapp/tmp/restart.txt
```

4.5. Rackup specifications for various web frameworks

This subsection shows example *config.ru* files for various web frameworks.

4.5.1. Camping

```
require 'rubygems'
require 'rack'
require 'camping'

##### Begin Camping application
Camping.goes :Blog

...your application code here...
##### End Camping application

run Rack::Adapter::Camping.new(Blog)
```

For Camping versions 2.0 and up, using `run Blog` as the final line will do.

4.5.2. Halcyon

```
require 'rubygems'
require 'halcyon'
$LOAD_PATH.unshift(Halcyon.root / 'lib')
Halcyon::Runner.load_config Halcyon.root/'config'/'config.yml'
run Halcyon::Runner.new
```

4.5.3. Mack

```
ENV["MACK_ENV"] = ENV["RACK_ENV"]
load("Rakefile")
require 'rubygems'
require 'mack'
run Mack::Utils::Server.build_app
```

4.5.4. Merb

```
require 'rubygems'
require 'merb-core'

Merb::Config.setup(
  :merb_root   => ::File.expand_path(::File.dirname(__FILE__)),
  :environment => ENV['RACK_ENV']
)
Merb.environment = Merb::Config[:environment]
Merb.root = Merb::Config[:merb_root]
Merb::BootLoader.run

run Merb::Rack::Application.new
```

4.5.5. Ramaze

```
require "rubygems"
require "ramaze"
Ramaze.trait[:essentials].delete Ramaze::Adapter
require "start"
Ramaze.start!
run Ramaze::Adapter::Base
```

4.5.6. Sinatra

```
require 'rubygems'
require 'sinatra'
require 'app.rb'

run Sinatra::Application
```

5. Configuring Phusion Passenger

After installation, Phusion Passenger does not need any further configurations. Nevertheless, the system administrator may be interested in changing Phusion Passenger's behavior. Phusion Passenger's Apache module supports the following configuration options:

5.1. PassengerRoot <directory>

The location to the Phusion Passenger root directory. This configuration option is essential to Phusion Passenger, and allows Phusion Passenger to locate its own data files. The correct value is given by the installer.

If you've moved Phusion Passenger to a different directory then you need to update this option

as well. Please read Moving Phusion Passenger to a different directory for more information.

This required option may only occur once, in the global server configuration.

5.2. PassengerRuby <filename>

This option allows one to specify the Ruby interpreter to use.

This option may only occur once, in the global server configuration. The default is *ruby*.

5.3. PassengerAppRoot <path/to/root>

By default, Phusion Passenger assumes that the application's root directory is the parent directory of the *public* directory. This option allows one to specify the application's root independently from the DocumentRoot, which is useful if the *public* directory lives in a non-standard place.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

In each place, it may be specified at most once.

Example:

```
<VirtualHost test.host>
    DocumentRoot /var/rails/zena/sites/example.com/public
    PassengerAppRoot /var/rails/zena   # <-- normally Phusion Passenger would
                                       #     have assumed that the application
                                       #     root is "/var/rails/zena/sites/exam
</VirtualHost>
```

5.4. PassengerSpawnMethod <string>

"What spawn method should I use?"

This subsection attempts to describe spawn methods, but it's okay if you don't (want to) understand it, as it's mostly a technical detail. You can basically follow this rule of thumb:

> If your application works on Mongrel, but not on Phusion Passenger, then set `PassengerSpawnMethod` to *conservative*. Otherwise, leave it at *smart-lv2* (the default).

However, we do recommend you to try to understand it. The *smart* and *smart-lv2* spawn methods bring many benefits.

Internally, Phusion Passenger spawns multiple Ruby application processes in order to handle requests. But there are multiple ways with which processes can be spawned, each having its own set of pros and cons. Supported spawn methods are:

smart

When this spawn method is used, Phusion Passenger will attempt to cache any framework code (e.g. Ruby on Rails itself) and application code for a limited period of time. Please read Spawning methods explained for a more detailed explanation of what smart spawning exactly does.

Pros: This can significantly decrease spawn time (by as much as 90%). And, when Ruby Enterprise Edition is used, memory usage can be reduced by 33% on average.

Cons: Some applications and libraries are not compatible with smart spawning. If that's the case for your application, then you should use *conservative* as spawning method. Please read Spawning methods explained for possible compatibility issues.

smart-lv2

This spawning method is similar to *smart* but it skips the framework spawner and uses the application spawner directly. This means the framework code is not cached between multiple applications, although it is still cached within instances of the same application. Please read Spawning methods explained for a more detailed explanation of what smart-lv2 spawning exactly does.

Pros: It is compatible with a larger number of applications when compared to the *smart* method, and still performs some caching.

Cons: It is slower than smart spawning if you have many applications which use the same framework version. It is therefore advised that shared hosts use the *smart* method instead.

conservative

This spawning method is similar to the one used in Mongrel Cluster. It does not perform any code caching at all. Please read Spawning methods explained for a more detailed explanation of what conservative spawning exactly does.

Pros: Conservative spawning is guaranteed to be compatible with all applications and libraries.

Cons: Much slower than smart spawning. Every spawn action will be equally slow, though no slower than the startup time of a single server in Mongrel Cluster. Conservative spawning will also render Ruby Enterprise Edition's memory reduction technology useless.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.

In each place, it may be specified at most once. The default value is *smart-lv2*.

5.5. PassengerUseGlobalQueue <on|off>

Turns the use of global queuing on or off.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.

In each place, it may be specified at most once. The default value is *on*.

This feature is sponsored by 37signals.

What does this option do?

Recall that Phusion Passenger spawns multiple backend processes (e.g. multiple Ruby on Rails processes), each which processes HTTP requests serially. One of Phusion Passenger's jobs is to forward HTTP requests to a suitable backend process. A backend process may take an arbitrary amount of time to process a specific HTTP request. If the websites are (temporarily)

under high load, and the backend processes cannot process the requests fast enough, then some requests may have to be queued.

If global queuing is turned off, then Phusion Passenger will use *fair load balancing*. This means that each backend process will have its own private queue. Phusion Passenger will forward an HTTP request to the backend process that has the least amount of requests in its queue.

If global queuing is turned on, then Phusion Passenger will use a global queue that's shared between all backend processes. If an HTTP request comes in, and all the backend processes are still busy, then Phusion Passenger will wait until at least one backend process is done, and will then forward the request to that process.

When to turn on global queuing?

You should turn on global queuing if one of your web applications may have long-running requests.

For example suppose that:

- global queuing is turned off.
- we're currently in a state where all backend processes have 3 requests in their queue, except for a single backend process, which has 1 request in its queue.

The situation looks like this:

```
Backend process A:  [*     ]  (1 request in queue)
Backend process B:  [***   ]  (3 requests in queue)
Backend process C:  [***   ]  (3 requests in queue)
Backend process D:  [***   ]  (3 requests in queue)
```

Each process is currently serving short-running requests.

Phusion Passenger will forward the next request to backend process A. A will now have 2 items in its queue. We'll mark this new request with an X:

```
Backend process A:  [*X    ]  (2 request in queue)
Backend process B:  [***   ]  (3 requests in queue)
Backend process C:  [***   ]  (3 requests in queue)
Backend process D:  [***   ]  (3 requests in queue)
```

Assuming that B, C and D still aren't done with their current request, the next HTTP request - let's call this Y - will be forwarded to backend process A as well, because it has the least number of items in its queue:

```
Backend process A:  [*XY   ]  (3 requests in queue)
Backend process B:  [***   ]  (3 requests in queue)
Backend process C:  [***   ]  (3 requests in queue)
Backend process D:  [***   ]  (3 requests in queue)
```

But if request X happens to be a long-running request that needs 60 seconds to complete, then we'll have a problem. Y won't be processed for at least 60 seconds. It would have been a better idea if Y was forward to processes B, C or D instead, because they only have short-living requests in their queues.

This problem will be avoided entirely if you turn global queuing on. With global queuing, all backend processes will share the same queue. The first backend process that becomes available will take from the queue, and so this "queuing-behind-long-running-request" problem will never occur.

You can set this option to *off* to completely disable Phusion Passenger for a certain location. This is useful if, for example, you want to integrate a PHP application into the same virtual host as a Rails application.

Suppose that you have a Rails application in */apps/foo*. Suppose that you've dropped Wordpress — a blogging application written in PHP — in */apps/foo/public/wordpress*. You can then configure Phusion Passenger as follows:

```
<VirtualHost *:80>
    ServerName www.foo.com
    DocumentRoot /apps/foo/public
    <Directory /apps/foo/public/wordpress>
        PassengerEnabled off
        AllowOverride all      # <-- Makes Wordpress's .htaccess file work.
    </Directory>
</VirtualHost>
```

This way, Phusion Passenger will not interfere with Wordpress.

PassengerEnabled may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*.

In each place, it may be specified at most once. The default value is *on*.

5.7. PassengerTempDir <directory>

Specifies the directory that Phusion Passenger should use for storing temporary files. This includes things such as Unix socket files, buffered file uploads (see also PassengerUploadBufferDir), etc.

This option may be specified once, in the global server configuration. The default temp directory that Phusion Passenger uses is */tmp*.

This option is especially useful if Apache is not allowed to write to /tmp (which is the case on some systems with strict SELinux policies) or if the partition that /tmp lives on doesn't have enough disk space.

Command line tools

Some Phusion Passenger command line administration tools, such as `passenger-status`, must know what Phusion Passenger's temp directory is in order to function properly. You can pass the directory through the `PASSENGER_TMPDIR` environment variable, or the `TMPDIR` environment variable (the former will be used if both are specified).

For example, if you set *PassengerTempDir* to */my_temp_dir*, then invoke `passenger-status` after you've set the `PASSENGER_TMPDIR` or `TMPDIR` environment variable, like this:

```
export PASSENGER_TMPDIR=/my_temp-dir
sudo -E passenger-status
# The -E option tells 'sudo' to preserve environment variables.
```

5.8. PassengerUploadBufferDir <directory>

Phusion Passenger buffers large file uploads to disk in order prevent slow file uploads from

blocking web applications. By default, a subdirectory in the system's temporary files directory (or a subdirectory in the directory specified in PassengerTempDir, if set) is automatically created for storing these buffered file uploads.

This configuration directive allows you to specify a different directory for storing buffered file uploads. If you've specified such a directory (as opposed to using Phusion Passenger's default) then you **must** ensure that this directory exists.

This configuration directive is also useful if you're using apache2-mpm-itk. The buffered file upload directory that Phusion Passenger creates by default has very strict permissions: it can only be accessed by the Apache worker processes. However, Phusion Passenger assumes that all Apache worker processes are running as the same user. apache2-mpm-itk breaks this assumption by running multiple Apache worker processes as different users. So if you're using apace2-mpm-itk, you should set this option to a directory that is writable by all Apache worker processes, such as */tmp*.

You may specify *PassengerUploadBufferDir* in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverrides Options` is enabled.

In each place, it may be specified at most once.

5.9. PassengerRestartDir <directory>

As described in the deployment chapters of this document, Phusion Passenger checks the file *tmp/restart.txt* in the applications' root directory for restarting applications. Sometimes it may be desirable for Phusion Passenger to look in a different directory instead, for example for security reasons (see below). This option allows you to customize the directory in which *restart.txt* is searched for.

You may specify *PassengerRestartDir* in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverrides Options` is enabled.

In each place, it may be specified at most once.

You can either set it to an absolute directory, or to a directory relative to the application root. Examples:

```
<VirtualHost *:80>
    ServerName www.foo.com
    # Phusion Passenger will check for /apps/foo/public/tmp/restart.txt
    DocumentRoot /apps/foo/public
</VirtualHost>

<VirtualHost *:80>
    ServerName www.bar.com
    DocumentRoot /apps/bar/public
    # An absolute filename is given; Phusion Passenger will
    # check for /restart_files/bar/restart.txt
    PassengerRestartDir /restart_files/bar
</VirtualHost>
```

```
<VirtualHost *:80>
    ServerName www.baz.com
    DocumentRoot /apps/baz/public
    # A relative filename is given; Phusion Passenger will
    # check for /apps/baz/restart_files/restart.txt
    #
    # Note that this directory is relative to the APPLICATION ROOT, *not*
    # the value of DocumentRoot!
    PassengerRestartDir restart_files
</VirtualHost>
```

What are the security reasons for wanting to customize PassengerRestartDir?

Touching restart.txt will cause Phusion Passenger to restart the application. So anybody who can touch restart.txt can effectively cause a Denial-of-Service attack by touching restart.txt over and over. If your web server or one of your web applications has the permission to touch restart.txt, and one of them has a security flaw which allows an attacker to touch restart.txt, then that will allow the attacker to cause a Denial-of-Service.

You can prevent this from happening by pointing PassengerRestartDir to a directory that's readable by Apache, but only writable by administrators.

5.10. Security options

5.10.1. PassengerUserSwitching <on|off>

Whether to enable user switching support.

This option may only occur once, in the global server configuration. The default value is *on*.

5.10.2. PassengerUser <username>

If user switching support is enabled, then Phusion Passenger will by default run the web application as the owner if the file *config/environment.rb* (for Rails apps) or *config.ru* (for Rack apps). This option allows you to override that behavior and explicitly set a user to run the web application as, regardless of the ownership of *environment.rb*/*config.ru*.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.

In each place, it may be specified at most once.

5.10.3. PassengerGroup <group name>

If user switching support is enabled, then Phusion Passenger will by default run the web application as the primary group of the owner of the file *config/environment.rb* (for Rails apps) or *config.ru* (for Rack apps). This option allows you to override that behavior and explicitly set a group to run the web application as, regardless of the ownership of *environment.rb*/*config.ru*.

<group name> may also be set to the special value *!STARTUP_FILE!*, in which case the web application's group will be set to *environment.rb*/*config.ru*'s group.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.

In each place, it may be specified at most once.

5.10.4. PassengerDefaultUser <username>

Phusion Passenger enables user switching support by default. This configuration option allows one to specify the user that applications must run as, if user switching fails or is disabled.

This option may only occur once, in the global server configuration. The default value is *nobody*.

5.10.5. PassengerDefaultGroup <group name>

Phusion Passenger enables user switching support by default. This configuration option allows one to specify the group that applications must run as, if user switching fails or is disabled.

This option may only occur once, in the global server configuration. The default value is the primary group of the user specifiied by PassengerDefaultUser.

5.10.6. PassengerFriendlyErrorPages <on|off>

Phusion Passenger can display friendly error pages whenever an application fails to start. This friendly error page presents the startup error message, some suggestions for solving the problem, and a backtrace. This feature is very useful during application development and useful for less experienced system administrators, but the page might reveal potentially sensitive information, depending on the application. Experienced system administrators who are using Phusion Passenger on serious production servers should consider turning this feature off.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

In each place, it may be specified at most once. The default value is *on*.

5.11. Resource control and optimization options

5.11.1. PassengerMaxPoolSize <integer>

The maximum number of Ruby on Rails or Rack application instances that may be simultaneously active. A larger number results in higher memory usage, but improved ability to handle concurrent HTTP clients.

The optimal value depends on your system's hardware and the server's average load. You should experiment with different values. But generally speaking, the value should be at least equal to the number of CPUs (or CPU cores) that you have. If your system has 2 GB of RAM, then we recommend a value of *30*. If your system is a Virtual Private Server (VPS) and has about 256 MB RAM, and is also running other services such as MySQL, then we recommend a value of *2*.

If you find that your server is unable to handle the load on your Rails/Rack websites (i.e. running out of memory) then you should lower this value. (Though if your sites are really that popular, then you should strongly consider upgrading your hardware or getting more servers.)

This option may only occur once, in the global server configuration. The default value is *6*.

We strongly recommend you to use Ruby Enterprise Edition. This allows you to reduce the memory usage of your Ruby on Rails applications by about 33%. And it's not hard to install.

5.11.2. PassengerMinInstances <integer>

This specifies the minimum number of application instances that must be kept around whenever Phusion Passenger cleans up idle instances. You should set this option to a non-zero value if you want to avoid potentially long startup times after a website has been idle for an extended period.

Please note that this option does **not** pre-start application instances during Apache startup. It just makes sure that when the application is first accessed:

1. at least the given number of instances will be spawned.
2. the given number of processes will be kept around even when instances are being idle cleaned (see PassengerPoolIdleTime).

If you want to pre-start application instances during Apache startup, then you should use the PassengerPreStart directive, possibly in combination with *PassengerMinInstances*. This behavior might seem counter-intuitive at first sight, but PassengerPreStart explains the rationale behind it.

For example, suppose that you have the following configuration:

```
PassengerMaxPoolSize 15
PassengerPoolIdleTime 10

<VirtualHost *:80>
    ServerName foobar.com
    DocumentRoot /webapps/foobar/public
    PassengerMinInstances 3
</VirtualHost>
```

When you start Apache, there are 0 application instances for *foobar.com*. Things will stay that way until someone visits *foobar.com*. Suppose that there is only 1 visitor. 1 application instance will be started immediately to serve the visitor, while 2 will be spawned in the background. After 10 seconds, when the idle timeout has been reached, these 3 application instances will not be cleaned up.

Now suppose that there's a sudden spike of traffic, and 100 users visit *foobar.com* simultanously. Phusion Passenger will start 12 more application instances. After the idle timeout of 10 seconds have passed, Phusion Passenger will clean up 12 application instances, keeping 3 instances around.

The PassengerMinInstances option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Limits` is on.

In each place, it may be specified at most once. The default value is *1*.

5.11.3. PassengerMaxInstancesPerApp <integer>

The maximum number of application instances that may be simultaneously active for a single application. This helps to make sure that a single application will not occupy all available slots in the application pool.

This value must be less than PassengerMaxPoolSize. A value of 0 means that there is no limit placed on the number of instances a single application may use, i.e. only the global limit of PassengerMaxPoolSize will be enforced.

This option may only occur once, in the global server configuration. The default value is *0*.

5.11.4. PassengerPoolIdleTime <integer>

The maximum number of seconds that an application instance may be idle. That is, if an application instance hasn't received any traffic after the given number of seconds, then it will be shutdown in order to conserve memory.

Decreasing this value means that applications will have to be spawned more often. Since spawning is a relatively slow operation, some visitors may notice a small delay when they visit your Rails/Rack website. However, it will also free up resources used by applications more quickly.

The optimal value depends on the average time that a visitor spends on a single Rails/Rack web page. We recommend a value of `2 * x`, where `x` is the average number of seconds that a visitor spends on a single Rails/Rack web page. But your mileage may vary.

When this value is set to *0*, application instances will not be shutdown unless it's really necessary, i.e. when Phusion Passenger is out of worker processes for a given application and one of the inactive application instances needs to make place for another application instance. Setting the value to 0 is recommended if you're on a non-shared host that's only running a few applications, each which must be available at all times.

This option may only occur once, in the global server configuration. The default value is *300*.

5.11.5. PassengerMaxRequests <integer>

The maximum number of requests an application instance will process. After serving that many requests, the application instance will be shut down and Phusion Passenger will restart it. A value of 0 means that there is no maximum: an application instance will thus be shut down when its idle timeout has been reached.

This option is useful if your application is leaking memory. By shutting it down after a certain number of requests, all of its memory is guaranteed to be freed by the operating system.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Limits` is on.

In each place, it may be specified at most once. The default value is *0*.

The PassengerMaxRequests directive should be considered as a workaround for misbehaving applications. It is advised that you fix the problem in your application rather than relying on these directives as a measure to avoid memory leaks.

5.11.6. PassengerStatThrottleRate <integer>

By default, Phusion Passenger performs several filesystem checks (or, in programmers jargon, *stat() calls*) each time a request is processed:

- It checks whether *config/environment.rb*, *config.ru* or *passenger_wsgi.py* is present, in order to autodetect Rails, Rack and WSGI applications.
- It checks whether *restart.txt* has changed or whether *always_restart.txt* exists, in order

to determine whether the application should be restarted.

On some systems where disk I/O is expensive, e.g. systems where the harddisk is already being heavily loaded, or systems where applications are stored on NFS shares, these filesystem checks can incur a lot of overhead.

You can decrease or almost entirely eliminate this overhead by setting *PassengerStatThrottleRate*. Setting this option to a value of *x* means that the above list of filesystem checks will be performed at most once every *x* seconds. Setting it to a value of *0* means that no throttling will take place, or in other words, that the above list of filesystem checks will be performed on every request.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Limits` is on.

In each place, it may be specified at most once. The default value is *0*.

5.11.7. PassengerPreStart <url>

By default, Phusion Passenger does not start any application instances until said web application is first accessed. The result is that the first visitor of said web application might experience a small delay as Phusion Passenger is starting the web application on demand. If that is undesirable, then this directive can be used to pre-started application instances during Apache startup.

A few things to be careful of:

- This directive accepts the **URL** of the web application you want to pre-start, not a on/off value! This might seem a bit weird, but read on for rationale. As for the specifics of the URL:
  - The domain part of the URL must be equal to the value of the *ServerName* directive of the VirtualHost block that defines the web application.
  - Unless the web application is deployed on port 80, the URL should contain the web application's port number too.
  - The path part of the URL must point to some URI that the web application handles.
- You will probably want to combine this option with PassengerMinInstances because application instances started with *PassengerPreStart* are subject to the usual idle timeout rules. See the example below for an explanation.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.

In each place, it may be specified any number of times.

Example 1: basic usage

Suppose that you have the following web applications.

```
<VirtualHost *:80>
   ServerName foo.com
   DocumentRoot /webapps/foo/public
</VirtualHost>

<VirtualHost *:3500>
```

```
   ServerName bar.com
   DocumentRoot /webapps/bar/public
</VirtualHost>
```

You want both of them to be pre-started during Apache startup. The URL for foo.com is *http://foo.com/* (or, equivalently, *http://foo.com:80/*) and the URL for bar.com is *http://bar.com:3500/*. So we add two PassengerPreStart directives, like this:

```
<VirtualHost *:80>
   ServerName foo.com
   DocumentRoot /webapps/foo/public
</VirtualHost>

<VirtualHost *:3500>
   ServerName bar.com
   DocumentRoot /webapps/bar/public
</VirtualHost>

PassengerPreStart http://foo.com/           # <--- added
PassengerPreStart http://bar.com:3500/      # <--- added
```

Example 2: pre-starting apps that are deployed in sub-URIs

Suppose that you have a web application deployed in a sub-URI */store*, like this:

```
<VirtualHost *:80>
   ServerName myblog.com
   DocumentRoot /webapps/wordpress
   RailsBaseURI /store
</VirtualHost>
```

Then specify the domain name of its containing virtual host followed by the sub-URI, like this:

```
<VirtualHost *:80>
   ServerName myblog.com
   DocumentRoot /webapps/wordpress
   RailsBaseURI /store
</VirtualHost>

PassengerPreStart http://myblog.com/store    # <----- added
```

The sub-URI **must** be included; if you don't then the directive will have no effect. The following example is wrong and won't pre-start the store web application:

```
PassengerPreStart http://myblog.com/    # <----- WRONG! Missing "/store" part.
```

Example 3: combining with PassengerMinInstances

Application instances started with PassengerPreStart are also subject to the idle timeout rules as specified by PassengerPoolIdleTime! That means that by default, the pre-started application instances for foo.com are bar.com are shut down after a few minutes of inactivity. If you don't want that to happen, then you should combine PassengerPreStart with PassengerMinInstances, like this:

```
<VirtualHost *:80>
   ServerName foo.com
   DocumentRoot /webapps/foo/public
   PassengerMinInstances 1      # <--- added
</VirtualHost>

<VirtualHost *:3500>
   ServerName bar.com
   DocumentRoot /webapps/bar/public
   PassengerMinInstances 1      # <--- added
</VirtualHost>

PassengerPreStart http://foo.com/
PassengerPreStart http://bar.com:3500/
```

So why a URL? Why not just an on/off flag?

A directive that accepts a simple on/off flag is definitely more intuitive, but due technical difficulties w.r.t. the way Apache works, it's very hard to implement it like that:

- It is very hard to obtain a full list of web applications defined in the Apache configuration file(s). In other words, it's hard for Phusion Passenger to know which web applications are deployed on Apache until a web application is first accessed, and without such a list Phusion Passenger wouldn't know which web applications to pre-start. It's probably not completely impossible to obtain such a list, but this brings us to the following point;
- Users expect things like *mod_env* to work even in combination with Phusion Passenger. For example some people put "SetEnv PATH=...." in their virtual host block and they expect the web application to pick that environment variable up when it's started. Information like this is stored in module-specific locations that Phusion Passenger cannot access directly. Even if the previous bullet point is solved and we can obtain a list of web applications, we cannot start the application with the correct mod_env information. mod_env is just one such example; there are probably many other Apache modules, all of which people expect to work, but we cannot answer to those expectations if PassengerPreStart is implemented as a simple on/off flag.

So as a compromise, we made it accept a URL. This is easier to implement for us and altough it looks weird, it behaves consistently w.r.t. cooperation with other Apache modules.

What does Phusion Passenger do with the URL?

During Apache startup, Phusion Passenger will send a dummy HEAD request to the given URL and discard the result. In other words, Phusion Passenger simulates a web access at the given URL. However this simulated request is always sent to localhost, **not** to the IP that the domain resolves to. Suppose that bar.com in example 1 resolves to 209.85.227.99; Phusion Passenger will send the following HTTP request to 127.0.0.1 port 3500 (and not to 209.85.227.99 port 3500):

```
HEAD / HTTP/1.1
Host: bar.com
Connection: close
```

Similarly, for example 2, Phusion Passenger will send the following HTTP request to 127.0.0.1 port 80:

```
HEAD /store HTTP/1.1
Host: myblog.com
```

```
Connection: close
```

Do I need to edit /etc/hosts and point the domain in the URL to 127.0.0.1?

No. See previous subsection.

My web application consists of multiple web servers. What URL do I need to specify, and in which web server's Apache config file?

Put the web application's virtual host's ServerName value and the virtual host's port in the URL, and put PassengerPreStart on all machines that you want to pre-start the web application on. The simulated web request is always sent to 127.0.0.1, with the domain name in the URL as value for the *Host* HTTP header, so you don't need to worry about the request ending up at a different web server in the cluster.

Does PassengerPreStart support https:// URLs?

Yes. And it does not perform any certificate validation.

5.11.8. PassengerHighPerformance <on|off>

By default, Phusion Passenger is compatible with mod_rewrite and most other Apache modules. However, a lot of effort is required in order to be compatible. If you turn *PassengerHighPerformance* to *on*, then Phusion Passenger will be a little faster, in return for reduced compatibility with other Apache modules.

In places where *PassengerHighPerformance* is turned on, mod_rewrite rules will likely not work. mod_autoindex (the module which displays a directory index) will also not work. Other Apache modules may or may not work, depending on what they exactly do. We recommend you to find out how other modules behave in high performance mode via testing.

This option is **not** an all-or-nothing global option: you can enable high performance mode for certain virtual hosts or certain URLs only. The *PassengerHighPerformance* option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*.

In each place, it may be specified at most once. The default value is *off*, so high performance mode is disabled by default, and you have to explicitly enable it.

When to enable high performance mode?

If you do not use mod_rewrite or other Apache modules then it might make sense to enable high performance mode.

It's likely that some of your applications depend on mod_rewrite or other Apache modules, while some do not. In that case you can enable high performance for only those applications that don't use other Apache modules. For example:

```
<VirtualHost *:80>
    ServerName www.foo.com
    DocumentRoot /apps/foo/public
    .... mod_rewrite rules or options for other Apache modules here ...
</VirtualHost>

<VirtualHost *:80>
    ServerName www.bar.com
    DocumentRoot /apps/bar/public
    PassengerHighPerformance on
```

```
</VirtualHost>
```

In the above example, high performance mode is only enabled for www.bar.com. It is disabled for everything else.

If your application generally depends on mod_rewrite or other Apache modules, but a certain URL that's accessed often doesn't depend on those other modules, then you can enable high performance mode for a certain URL only. For example:

```
<VirtualHost *:80>
    ServerName www.foo.com
    DocumentRoot /apps/foo/public
    .... mod_rewrite rules or options for other Apache modules here ...

    <Location /chatroom/ajax_update_poll>
        PassengerHighPerformance on
    </Location>
</VirtualHost>
```

This enables high performance mode for http://www.foo.com/chatroom/ajax_update_poll only.

5.12. Compatibility options

5.12.1. PassengerResolveSymlinksInDocumentRoot <on|off>

Configures whether Phusion Passenger should resolve symlinks in the document root. Please refer to How Phusion Passenger detects whether a virtual host is a web application for more information.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

In each place, it may be specified at most once. It is off by default.

5.12.2. PassengerAllowEncodedSlashes <on|off>

By default, Apache doesn't support URLs with encoded slashes (%2f), e.g. URLs like this: `/users/fujikura%2fyuu`. If you access such an URL then Apache will return a 404 Not Found error. This can be solved by turning on PassengerAllowEncodedSlashes as well as Apache's AllowEncodedSlashes.

Is it important that you turn on both AllowEncodedSlashes **and** PassengerAllowEncodedSlashes, otherwise this feature will not work properly.

PassengerAllowEncodedSlashes may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

In each place, it may be specified at most once. It is off by default.

Please note however that turning on support for encoded slashes will break support for mod_rewrite passthrough rules. Because of bugs/limitations in Apache, Phusion Passenger can

support either encoded slashes or mod_rewrite passthrough rules, but not both at the same time. Luckily this option can be specified anywhere, so you can enable it only for virtual hosts or URLs that need it:

```
<VirtualHost *:80>
    ServerName www.example.com
    DocumentRoot /webapps/example/public
    AllowEncodedSlashes on
    RewriteEngine on

    # Check for maintenance file and redirect all requests
    RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
    RewriteCond %{SCRIPT_FILENAME} !maintenance.html
    RewriteRule ^.*$ /system/maintenance.html [L]

    # Make /about an alias for /info/about.
    RewriteRule ^/about$ /info/about [PT,L]

    <Location ~ "^/users/">
        # In a location block so that it doesn't interfere with the
        # above /about mod_rewrite rule.
        PassengerAllowEncodedSlashes on
    </Location>
</VirtualHost>
```

With this, http://www.example.com/users/fujikura%2fyuu will work properly, and accessing http://www.example.com/about will properly display the result of http://www.example.com/info/about. Notice that PassengerAllowEncodedSlashes only interferes with passthrough rules, not with any other mod_rewrite rules. The rules for displaying maintenance.html will work fine even for URLs starting with "/users".

5.13. Logging and debugging options

5.13.1. PassengerLogLevel <integer>

This option allows one to specify how much information Phusion Passenger should write to the Apache error log file. A higher log level value means that more information will be logged.

Possible values are:

- *0*: Show only errors and warnings.
- *1*: Show the most important debugging information. This might be useful for system administrators who are trying to figure out the cause of a problem.
- *2*: Show more debugging information. This is typically only useful for developers.
- *3*: Show even more debugging information.

This option may only occur once, in the global server configuration. The default is *0*.

5.13.2. PassengerDebugLogFile <filename>

By default Phusion Passenger debugging and error messages are written to the global web server error log. This option allows one to specify the file that debugging and error messages should be written to instead.

This option may only occur once, in the global server configuration.

5.14. Ruby on Rails-specific options

5.14.1. RailsAutoDetect <on|off>

Whether Phusion Passenger should automatically detect whether a virtual host's document root is a Ruby on Rails application. The default is *on*.

This option may occur in the global server configuration or in a virtual host configuration block.

For example, consider the following configuration:

```
RailsAutoDetect off
<VirtualHost *:80>
    ServerName www.mycook.com
    DocumentRoot /webapps/mycook/public
</VirtualHost>
```

If one goes to *http://www.mycook.com/*, the visitor will see the contents of the */webapps/mycook/public* folder, instead of the output of the Ruby on Rails application.

It is possible to explicitly specify that the host is a Ruby on Rails application by using the RailsBaseURI configuration option:

```
RailsAutoDetect off
<VirtualHost *:80>
    ServerName www.mycook.com
    DocumentRoot /webapps/mycook/public
    RailsBaseURI /          # This line has been added.
</VirtualHost>
```

5.14.2. RailsBaseURI <uri>

Used to specify that the given URI is a Rails application. See Deploying Rails to a sub URI for an example.

It is allowed to specify this option multiple times. Do this to deploy multiple Rails applications in different sub-URIs under the same virtual host.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

5.14.3. RailsEnv <string>

This option allows one to specify the default `RAILS_ENV` value.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

In each place, it may be specified at most once. The default value is *production*.

5.14.4. RailsFrameworkSpawnerIdleTime <integer>

The FrameworkSpawner server (explained in Spawning methods explained) has an idle

timeout, just like the backend processes spawned by Phusion Passenger do. That is, it will automatically shutdown if it hasn't done anything for a given period.

This option allows you to set the FrameworkSpawner server's idle timeout, in seconds. A value of *0* means that it should never idle timeout.

Setting a higher value will mean that the FrameworkSpawner server is kept around longer, which may slightly increase memory usage. But as long as the FrameworkSpawner server is running, the time to spawn a Ruby on Rails backend process only takes about 40% of the time that is normally needed, assuming that you're using the *smart* spawning method. So if your system has enough memory, is it recommended that you set this option to a high value or to *0*.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.

In each place, it may be specified at most once. The default value is *1800* (30 minutes).

5.14.5. RailsAppSpawnerIdleTime <integer>

The ApplicationSpawner server (explained in Spawning methods explained) has an idle timeout, just like the backend processes spawned by Phusion Passenger do. That is, it will automatically shutdown if it hasn't done anything for a given period.

This option allows you to set the ApplicationSpawner server's idle timeout, in seconds. A value of *0* means that it should never idle timeout.

Setting a higher value will mean that the ApplicationSpawner server is kept around longer, which may slightly increase memory usage. But as long as the ApplicationSpawner server is running, the time to spawn a Ruby on Rails backend process only takes about 10% of the time that is normally needed, assuming that you're using the *smart* or *smart-lv2* spawning method. So if your system has enough memory, is it recommended that you set this option to a high value or to *0*.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.

In each place, it may be specified at most once. The default value is *600* (10 minutes).

5.15. Rack-specific options

5.15.1. RackAutoDetect <on|off>

Whether Phusion Passenger should automatically detect whether a virtual host's document root is a Rack application. The default is *on*.

This option may occur in the global server configuration or in a virtual host configuration block.

For example, consider the following configuration:

```
RackAutoDetect off
<VirtualHost *:80>
    ServerName www.rackapp.com
    DocumentRoot /webapps/my_rack_app/public
</VirtualHost>
```

If one goes to *http://www.rackapp.com/*, the visitor will see the contents of the */webapps/my_rack_app/public* folder, instead of the output of the Rack application.

It is possible to explicitly specify that the host is a Rack application by using the RackBaseURI

configuration option:

```
RackAutoDetect off
<VirtualHost *:80>
    ServerName www.rackapp.com
    DocumentRoot /webapps/my_rack_app/public
    RackBaseURI /       # This line was added
</VirtualHost>
```

5.15.2. RackBaseURI <uri>

Used to specify that the given URI is a Rack application. See Deploying Rack to a sub URI for an example.

It is allowed to specify this option multiple times. Do this to deploy multiple Rack applications in different sub-URIs under the same virtual host.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

5.15.3. RackEnv <string>

The given value will be accessible in Rack applications in the `RACK_ENV` environment variable. This allows one to define the environment in which Rack applications are run, very similar to `RAILS_ENV`.

This option may occur in the following places:

- In the global server configuration.
- In a virtual host configuration block.
- In a `<Directory>` or `<Location>` block.
- In *.htaccess*, if `AllowOverride Options` is on.

In each place, it may be specified at most once. The default value is *production*.

5.16. Deprecated options

The following options have been deprecated, but are still supported for backwards compatibility reasons.

5.16.1. RailsRuby

Deprecated in favor of PassengerRuby.

5.16.2. RailsUserSwitching

Deprecated in favor of PassengerUserSwitching.

5.16.3. RailsDefaultUser

Deprecated in favor of PassengerDefaultUser.

5.16.4. RailsAllowModRewrite

This option doesn't do anything anymore in recent versions of Phusion Passenger.

5.16.5. RailsSpawnMethod

Deprecated in favor of PassengerSpawnMethod.

6. Troubleshooting

6.1. Operating system-specific problems

6.1.1. MacOS X: The installer cannot locate MAMP's Apache

Symptoms

The installer finds Apache 2 development headers at `/Applications/MAMP/Library/bin/apxs`. However, Apache cannot be found. The installer also outputs the following error:

```
cannot open /Applications/MAMP/Library/build/config_vars.mk:
No such file or directory at /Applications/MAMP/Library/bin/apxs line 218.
```

Your MAMP installation seems to be broken. In particular, *config_vars.mk* is missing. Please read this forum topic to learn how to fix this problem.

See also this bug report.

6.2. Problems during installation

6.2.1. Ruby development headers aren't installed

Symptoms

Installing Phusion Passenger fails because of one of the following errors:

- The Phusion Passenger installer tells you that the Ruby development headers aren't installed.
- The error message "'no such file to load — mkmf'" occurs.
- The error message "'ruby.h: No such file or directory'" occurs.

Phusion Passenger makes use of a native extension, so the Ruby development headers must be installed. On most Linux systems, Ruby and the Ruby development headers are contained in separate packages, so having Ruby installed does not automatically imply having the development headers installed.

Here's how you can install the development headers:

Ubuntu/Debian
: Please type:

```
sudo apt-get install ruby1.8-dev
```

Fedora/CentOS/RHEL
: Please type:

```
su -c 'yum install ruby-devel'
```

FreeBSD
: Please install Ruby from *ports* or with `pkg_add`. If that fails, please install Ruby from source.

MacOS X
: Please install Ruby from source.

Other operating systems
: Please consult your operating system's native package database. There should be a package containing the Ruby development headers. If that fails, please install Ruby from source.

If you've installed a new Ruby version (i.e. your system now contains multiple Ruby installations), then you will need to tell Phusion Passenger which Ruby installation you want to use. Please read Specifying the correct Ruby installation.

6.2.2. Apache development headers aren't installed

Symptoms

Installing Phusion Passenger fails because of one of the following errors:

- The installer says that the Apache development headers aren't installed.
- The error message "'httpd.h: No such file or directory"' occurs.

  (Instead of *httpd.h*, the message might also be *http_config.h* or something else similar to *http_*.h*.)

Ubuntu
: Please type:

```
sudo apt-get install apache2-prefork-dev
```

Debian
: Please type:

```
sudo apt-get install apache2-dev
```

Fedora/CentOS/RHEL
: Please type:

```
su -c 'yum install httpd-devel'
```

FreeBSD
: Please install Apache from *ports* or with `pkg_add`. If that fails, please install Apache from source.

MacOS X
: Please install Apache from source.

Other operating systems
: Please consult your operating system's native package database. There should be a package containing the Apache development headers. If that fails, please install Apache

from source.

6.2.3. APR development headers aren't installed

Symptoms

Installing Phusion Passenger fails because one of the following errors:

- The installer tells you that APR development headers aren't installed.
- The error message "'apr_pools.h: No such file or directory"' occurs.
- The error message "'apr_strings.h: No such file or directory"' occurs.

Ubuntu

Please type:

```
sudo apt-get install libapr1-dev
```

Debian

Please type:

```
sudo apt-get install libapr1-dev
```

Fedora/CentOS/RHEL

Please type:

```
su -c 'yum install apr-devel'
```

Other Linux distributions

Please consult your distribution's package database. There should be a package which provides APR development headers.

Other operating systems

The APR development are bundled with Apache. If the APR headers aren't, then it probably means that they have been removed after Apache's been installed. Please reinstall Apache to get back the APR headers.

6.2.4. Phusion Passenger is using the wrong Apache during installation

Please Specifying the correct Apache installation, and re-run the Phusion Passenger installer.

6.2.5. Phusion Passenger is using the wrong Ruby during installation

Please Specifying the correct Ruby installation, and re-run the Phusion Passenger installer.

6.3. Problems after installation

The golden tip: read your Apache error logs!

mod_passenger will write all errors to the Apache error log. So if you're experiencing post-installation problems, please look inside the Apache error logs. It will tell you what exactly went wrong.

6.3.1. My Rails application works on Mongrel, but not on Phusion Passenger

Please try setting PassengerSpawnMethod to *conservative*.

6.3.2. Phusion Passenger has been compiled against the wrong Apache installation

Symptoms

Apache crashes during startup (after being daemonized). The Apache error log says "'seg fault or similar nasty error detected in the parent process"'.

This problem is most likely to occur on MacOS X. Most OS X users have multiple Apache installations on their system.

To solve this problem, please specify the correct Apache installation, and reinstall Phusion Passenger.

6.3.3. I get a "304 Forbidden" error

See next subsection.

6.3.4. Static assets such as images and stylesheets aren't being displayed

Static assets are accelerated, i.e. they are served directly by Apache and do not go through the Rails stack. There are two reasons why Apache doesn't serve static assets correctly:

1. Your Apache configuration is too strict, and does not allow HTTP clients to access static assets. This can be achieved with an `Allow from all` directive in the correct place. For example:

   ```
   <Directory "/webapps/mycook/public">
      Options FollowSymLinks
      AllowOverride None
      Order allow,deny
      Allow from all
   </Directory>
   ```

   See also this discussion.

2. The Apache process doesn't have permission to access your Rails application's folder. Please make sure that the Rails application's folder, as well as all of its parent folders, have the correct permissions and/or ownerships.

6.3.5. The Apache error log says that the spawn manager script does not exist, or that it does not have permission to execute it

If you are sure that the *PassengerRoot* configuration option is set correctly, then this problem is most likely caused by the fact that you're running Apache with SELinux. On Fedora, CentOS and RedHat Enterprise Linux, Apache is locked down by SELinux policies.

To solve this problem, you must set some permissions on the Phusion Passenger files and folders, so that Apache can access them.

- If you've installed Phusion Passenger via a gem, then run this command to determine Phusion Passenger's root folder:

  ```
  passenger-config --root
  ```

  Next, run the following command:

```
chcon -R -h -t httpd_sys_content_t /path-to-passenger-root
```

where */path-to-passenger-root* should be replaced with whatever `passenger-config --root` printed.

- If you've installed Phusion Passenger via the source tarball, then run the following command:

```
chcon -R -h -t httpd_sys_content_t /path/to/passenger/folder
```

Once the permissions are fixed, restart Apache.

6.3.6. The Rails application reports that it's unable to start because of a permission error

Please check whether your Rails application's folder has the correct permissions. By default, Rails applications are started as the owner of the file *config/environment.rb*, except if the file is owned by root. If the file is owned by root, then the Rails application will be started as *nobody* (or as the user specify by RailsDefaultUser, if that's specified).

Please read User switching (security) for details.

6.3.7. My Rails application's log file is not being written to

There are a couple things that you should be aware of:

- By default, Phusion Passenger runs Rails applications in *production* mode, so please be sure to check *production.log* instead of *development.log*. See RailsEnv for configuration.
- By default, Phusion Passenger runs Rails applications as the owner of *environment.rb*. So the log file can only be written to if that user has write permission to the log file. Please *chmod* or *chown* your log file accordingly.

  See User switching (security) for details.

If you're using a RedHat-derived Linux distribution (such as Fedora or CentOS) then it is possible that SELinux is interfering. RedHat's SELinux policy only allows Apache to read/write directories that have the *httpd_sys_content_t* security context. Please run the following command to give your Rails application folder that context:

```
chcon -R -h -t httpd_sys_content_t /path/to/your/rails/app
```

6.4. Conflicting Apache modules

6.4.1. mod_userdir

mod_userdir is not compatible with Phusion Passenger at the moment.

6.4.2. MultiViews (mod_negotiation)

MultiViews is not compatible with Phusion Passenger. You should disable MultiViews for all Phusion Passenger hosts.

6.4.3. VirtualDocumentRoot

VirtualDocumentRoot is not compatible with Phusion Passenger at the moment.

7. Analysis and system maintenance

Phusion Passenger provides a set of tools, which are useful for system analysis, maintenance and troubleshooting.

7.1. Inspecting memory usage

Process inspection tools such as `ps` and `top` are useful, but they rarely show the correct memory usage. The real memory usage is usually lower than what `ps` and `top` report.

There are many technical reasons why this is so, but an explanation is beyond the scope of this Users Guide. We kindly refer the interested reader to operating systems literature about *virtual memory* and *copy-on-write*.

The tool `passenger-memory-stats` allows one to easily analyze Phusion Passenger's and Apache's real memory usage. For example:

```
[bash@localhost root]# passenger-memory-stats
------------ Apache processes --------------.
PID    PPID  Threads  VMSize   Private  Name
--------------------------------------------.
5947   1     9        90.6 MB  0.5 MB   /usr/sbin/apache2 -k start
5948   5947  1        18.9 MB  0.7 MB   /usr/sbin/fcgi-pm -k start
6029   5947  1        42.7 MB  0.5 MB   /usr/sbin/apache2 -k start
6030   5947  1        42.7 MB  0.5 MB   /usr/sbin/apache2 -k start
6031   5947  1        42.5 MB  0.3 MB   /usr/sbin/apache2 -k start
6033   5947  1        42.5 MB  0.4 MB   /usr/sbin/apache2 -k start
6034   5947  1        50.5 MB  0.4 MB   /usr/sbin/apache2 -k start
23482  5947  1        82.6 MB  0.4 MB   /usr/sbin/apache2 -k start
### Processes: 8
### Total private dirty RSS: 3.50 MB

--------- Passenger processes ---------.
PID    Threads  VMSize   Private  Name
---------------------------------------.
6026   1        10.9 MB  4.7 MB   Passenger spawn server
23481  1        26.7 MB  3.0 MB   Passenger FrameworkSpawner: 2.0.2
23791  1        26.8 MB  2.9 MB   Passenger ApplicationSpawner: /var/www/project
23793  1        26.9 MB  17.1 MB  Rails: /var/www/projects/app1-foobar
### Processes: 4
### Total private dirty RSS: 27.76 M
```

The *Private* or *private dirty RSS* field shows the **real** memory usage of processes. Here, we see that all the Apache worker processes only take less than 1 MB memory each. This is a lot less than the 50 MB-ish memory usage as shown in the *VMSize* column (which is what a lot of people think is the real memory usage, but is actually not).

Private dirty RSS reporting only works on Linux. Unfortunately other operating systems don't provide facilities for determining processes' private dirty RSS. On non-Linux systems, the Resident Set Size is reported instead.

7.2. Inspecting Phusion Passenger's internal status

One can inspect Phusion Passenger's internal status with the tool `passenger-status`. This tool must typically be run as root. For example:

```
[bash@localhost root]# passenger-status
----------- General information -----------
max      = 6
count    = 1
active   = 0
inactive = 1

----------- Domains -----------
/var/www/projects/app1-foobar:
  PID: 9617      Sessions: 0    Processed: 7       Uptime: 2m 23s
```

The *general information* section shows the following information:

max
: The maximum number of application instances that Phusion Passenger will spawn. This equals the value given for PassengerMaxPoolSize (Apache) or passenger_max_pool_size (Nginx).

count
: The number of application instances that are currently alive. This value is always less than or equal to *max*.

active
: The number of application instances that are currently processing requests. This value is always less than or equal to *count*.

inactive
: The number of application instances that are currently **not** processing requests, i.e. are idle. Idle application instances will be shutdown after a while, as can be specified with PassengerPoolIdleTime (Apache)/passenger_pool_idle_time (Nginx) (unless this value is set to 0, in which case application instances are never shut down via idle time). The value of *inactive* equals `count - active`.

The *domains* section shows, for each application directory, information about running application instances:

Sessions
: Shows how many HTTP client are currently in the queue of that application Instance, waiting to be processed.

Processed
: Indicates how many requests the instance has served until now. **Tip:** it's possible to limit this number with the PassengerMaxRequests configuration directive.

Uptime
: Shows for how long the application instance has been running.

Since Phusion Passenger uses fair load balancing by default, the number of sessions for the application instances should be fairly close to each other. For example, this is fairly normal:

```
  PID: 4281      Sessions: 2      Processed: 7      Uptime: 5m 11s
  PID: 4268      Sessions: 0      Processed: 5      Uptime: 4m 52s
  PID: 4265      Sessions: 1      Processed: 6      Uptime: 5m 38s
  PID: 4275      Sessions: 1      Processed: 7      Uptime: 3m 14s
```

But if you see a "spike", i.e. an application instance has an unusually high number of sessions compared to the others, then there might be a problem:

```
  PID: 4281      Sessions: 2      Processed: 7      Uptime: 5m 11s
```

```
  PID: 17468     Sessions: 8 <-+  Processed: 2      Uptime: 4m 47s
  PID: 4265      Sessions: 1   |  Processed: 6      Uptime: 5m 38s
  PID: 4275      Sessions: 1   |  Processed: 7      Uptime: 3m 14s
                               |
                              +---- "spike"
```

Possible reasons why spikes can occur:

1. Your application is busy processing a request that takes a very long time. If this is the case, then you might want to turn global queuing on.
2. Your application is frozen, i.e. has stopped responding. See Debugging frozen applications for tips.

7.3. Debugging frozen applications

If one of your application instances is frozen (stopped responding), then you can figure out where it is frozen by killing it with *SIGABRT*. This will cause the application to raise an exception, with a backtrace.

The exception (with full backtrace information) is normally logged into the Apache error log. But if your application or if its web framework has its own exception logging routines, then exceptions might be logged into the application's log files instead. This is the case with Ruby on Rails. So if you kill a Ruby on Rails application with *SIGABRT*, please check the application's *production.log* first (assuming that you're running it in a *production* environment). If you don't see a backtrace there, check the Apache error log.

It is safe to kill application instances, even in live environments. Phusion Passenger will restart killed application instances, as if nothing bad happened.

7.4. Accessing individual application processes

When a request is sent to the web server, Phusion Passenger will automatically forward the request to the most suitable application process, but sometimes it is desirable to be able to directly access the individual application processes. Use cases include, but are not limited to:

- One wants to debug a memory leak or memory bloat problem that only seems to appear on certain URIs. One can send a request to a specific process to see whether that request causes the process's memory usage to rise.
- The application caches data in local memory, and one wants to tell a specific application process to clear that local data.
- Other debugging use cases.

All individual application processes are accessible via HTTP, so you can use standard HTTP tools like *curl*. The exact addresses can be obtained with the command `passenger-status --verbose`. These sockets are all bound to 127.0.0.1, but the port number is dynamically assigned. As a security measure, the sockets are also protected with a process-specific random password, which you can see in the `passenger-status --verbose` output. This password must be sent through the "X-Passenger-Connect-Password" HTTP header.

Example:

```
bash# passenger-status --verbose
----------- General information -----------
max      = 6
```

```
count    = 2
active   = 0
inactive = 2
Waiting on global queue: 0

----------- Application groups -----------
/Users/hongli/Sites/rack.test:
  App root: /Users/hongli/Sites/rack.test
  * PID: 24235   Sessions: 0    Processed: 7       Uptime: 17s
      URL     : http://127.0.0.1:58122
      Password: nFfVOX1F8LjZ90HJh28Sd_htJOsgRsNne2QXKf8NIXw
  * PID: 24250   Sessions: 0    Processed: 4       Uptime: 1s
      URL     : http://127.0.0.1:57933
      Password: _RGXlQ9EGDGJKLevQ_qflUtF1KmxEo2UiRzMwIE1sBY
```

Here we see that the web application *rack.test* has two processes. Process 24235 is accessible via http://127.0.0.1:58122, and process 24250 is accessible via http://127.0.0.1:57933.

To access 24235 we must send its password, *nFfVOX1F8LjZ90HJh28Sd_htJOsgRsNne2QXKf8NIXw*, through the *X-Passenger-Connect-Password* HTTP header, like this:

```
bash# curl -H "X-Passenger-Connect-Password: nFfVOX1F8LjZ90HJh28Sd_htJOsgRsNne2Q
```

8. Tips

8.1. User switching (security)

There is a problem that plagues most PHP web hosts, namely the fact that all PHP applications are run in the same user context as the web server. So for example, Joe's PHP application will be able to read Jane's PHP application's passwords. This is obviously undesirable on many servers.

Phusion Passenger solves this problem by implementing *user switching*. A Rails application is started as the owner of the file *config/environment.rb*, and a Rack application is started as the owner of the file *config.ru*. So if */home/webapps/foo/config/environment.rb* is owned by *joe*, then Phusion Passenger will launch the corresponding Rails application as *joe* as well.

This behavior is the default, and you don't need to configure anything. But there are things that you should keep in mind:

- The owner of *environment.rb*/*config.ru* must have read access to the application's root directory, and read/write access to the application's *logs* directory.
- This feature is only available if Apache is started by *root*. This is the case on most Apache installations.
- Under no circumstances will applications be run as *root*. If *environment.rb*/*config.ru* is owned as root or by an unknown user, then the Rails/Rack application will run as the user specified by PassengerDefaultUser and PassengerDefaultGroup.

User switching can be disabled with the PassengerUserSwitching option.

8.2. Reducing memory consumption of Ruby on Rails applications by 33%

Is it possible to reduce memory consumption of your Rails applications by 33% on average, by

using Ruby Enterprise Edition. Please visit the website for details.

Note that this feature does not apply to Rack applications.

8.3. Capistrano recipe

Phusion Passenger can be combined with Capistrano. The following Capistrano recipe demonstrates Phusion Passenger support. It assumes that you're using Git as version control system.

```
set :application, "myapp"
set :domain,      "example.com"
set :repository,  "ssh://#{domain}/path-to-your-git-repo/#{application}.git"
set :use_sudo,    false
set :deploy_to,   "/path-to-your-web-app-directory/#{application}"
set :scm,         "git"

role :app, domain
role :web, domain
role :db,  domain, :primary => true

namespace :deploy do
  task :start, :roles => :app do
    run "touch #{current_release}/tmp/restart.txt"
  end

  task :stop, :roles => :app do
    # Do nothing.
  end

  desc "Restart Application"
  task :restart, :roles => :app do
    run "touch #{current_release}/tmp/restart.txt"
  end
end
```

8.4. Bundler support

Phusion Passenger has automatic support for Bundler. It works as follows:

- If you have a *.bundle/environment.rb* in your application root, then Phusion Passenger will require that file before loading your application.
- Otherwise, if you have a *Gemfile*, then Phusion Passenger will automatically call `Bundler.setup()` before loading your application.

It's possible that your application also calls `Bundler.setup` during loading, e.g. in *config.ru* or in *config/boot.rb*. This is the case with Rails 3, and is also the case if you modified your *config/boot.rb* according to the Bundler Rails 2.3 instructions. This leads to `Bundler.setup` being called twice, once before the application startup file is required and once during application startup. However this is harmless and doesn't have any negative effects.

Phusion Passenger assumes that you're using Bundler >= 0.9.5. If you don't want Phusion Passenger to run its Bundler support code, e.g. because you need to use an older version of Bundler with an incompatible API or because you use a system other than Bundler, then you can override Phusion Passenger's Bundler support code by creating a file *config/setup_load_paths.rb*. If this file exists then it will be required before loading the application startup file. In this file you can do whatever you need to setup Bundler or a

similar system.

8.5. Moving Phusion Passenger to a different directory

It is possible to relocate the Phusion Passenger files to a different directory. It involves two steps:

1. Moving the directory.
2. Updating the "PassengerRoot" configuration option in Apache.

For example, if Phusion Passenger is located in */opt/passenger/*, and you'd like to move it to */usr/local/passenger/*, then do this:

1. Run the following command:

```
mv /opt/passenger /usr/local/passenger
```

2. Edit your Apache configuration file, and set:

```
PassengerRoot /usr/local/passenger
```

8.6. Installing multiple Ruby on Rails versions

Each Ruby on Rails applications that are going to be deployed may require a specific Ruby on Rails version. You can install a specific version with this command:

```
gem install rails -v X.X.X
```

where *X.X.X* is the version number of Ruby on Rails.

All of these versions will exist in parallel, and will not conflict with each other. Phusion Passenger will automatically make use of the correct version.

8.7. Making the application restart after each request

In some situations it might be desirable to restart the web application after each request, for example when developing a non-Rails application that doesn't support code reloading, or when developing a web framework.

To achieve this, simply create the file *tmp/always_restart.txt* in your application's root folder. Unlike *restart.txt*, Phusion Passenger does not check for this file's timestamp: Phusion Passenger will always restart the application, as long as *always_restart.txt* exists.

If you're just developing a Rails application then you probably don't need this feature. If you set *RailsEnv development* in your Apache configuration, then Rails will automatically reload your application code after each request. *always_restart.txt* is only useful if you're working on Ruby on Rails itself, or when you're not developing a Rails application and your web framework does not support code reloading.

8.8. How to fix broken images/CSS/JavaScript URIs in sub-URI deployments

Some people experience broken images and other broken static assets when they deploy their application to a sub-URI (i.e. *http://mysite.com/railsapp/*). The reason for this usually is that you used a static URI for your image in the views. This means your *img* source probably refers to something like */images/foo.jpg*. The leading slash means that it's an absolute URI: you're telling the browser to always load *http://mysite.com/images/foo.jpg* no matter what. The problem is that the image is actually at *http://mysite.com/railsapp/images/foo.jpg*. There are two ways to fix this.

The first way (not recommended) is to change your view templates to refer to *images/foo.jpg*. This is a relative URI: note the lack of a leading slash). What this does is making the path relative to the current URI. The problem is that if you use restful URIs, then your images will probably break again when you add a level to the URI. For example, when you're at *http://mysite.com/railsapp* the browser will look for *http://mysite.com/railsapp/images/foo.jpg*. But when you're at *http://mysite.com/railsapp/controller*. the browser will look for *http://mysite.com/railsapp/controller/images/foo.jpg*. So relative URIs usually don't work well with layout templates.

The second and highly recommended way is to always use Rails helper methods to output tags for static assets. These helper methods automatically take care of prepending the base URI that you've deployed the application to. For images there is `image_tag`, for JavaScript there is `javascript_include_tag` and for CSS there is `stylesheet_link_tag`. In the above example you would simply remove the *<img>* HTML tag and replace it with inline Ruby like this:

```
<%= image_tag("foo.jpg") %>
```

This will generate the proper image tag to `$RAILS_ROOT/public/images/foo.jpg` so that your images will always work no matter what sub-URI you've deployed to.

These helper methods are more valuable than you may think. For example they also append a timestamp to the URI to better facilitate HTTP caching. For more information, please refer to the Rails API docs.

8.9. X-Sendfile support

Phusion Passenger does not provide X-Sendfile support by itself. Please install mod_xsendfile for X-Sendfile support.

8.10. Upload progress

Phusion Passenger does not provide upload progress support by itself. Please try drogus's Apache upload progress module instead.

9. Under the hood

Phusion Passenger hides a lot of complexity for the end user (i.e. the web server system administrator), but sometimes it is desirable to know what is going on. This section describes a few things that Phusion Passenger does under the hood.

9.1. Static assets serving

Phusion Passenger accelerates serving of static files. This means that, if an URI maps to a file that exists, then Phusion Passenger will let Apache serve that file directly, without hitting the web application.

Phusion Passenger does all this without the need for any mod_rewrite rules. People who are switching from an old Mongrel-based setup might have mod_rewrite rules such as these:

```
# Check whether this request has a corresponding file; if that
# exists, let Apache serve it, otherwise forward the request to
# Mongrel.
RewriteCond %{DOCUMENT_ROOT}/%{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ balancer://mongrel%{REQUEST_URI} [P,QSA,L]
```

These kind of mod_rewrite rules are no longer required, and you can safely remove them.

9.2. Page caching support

For each HTTP request, Phusion Passenger will automatically look for a corresponding page cache file, and serve that if it exists. It does this by appending ".html" to the filename that the URI normally maps to, and checking whether that file exists. This check occurs after checking whether the original mapped filename exists (as part of static asset serving). All this is done without the need for special mod_rewrite rules.

For example, suppose that the browser requests */foo/bar*.

1. Phusion Passenger will first check whether this URI maps to a static file, i.e. whether the file */foo/bar* exists in the web application's *public* directory. If it does then Phusion Passenger will serve this file through Apache immediately.
2. If that doesn't exist, then Phusion Passenger will check whether the file */foo/bar.html* exists. If it does then Phusion Passenger will serve this file through Apache immediately.
3. If */foo/bar.html* doesn't exist either, then Phusion Passenger will forward the request to the underlying web application.

Note that Phusion Passenger's page caching support doesn't work if your web application uses a non-standard page cache directory, i.e. if it doesn't cache to the *public* directory. In that case you'll need to use mod_rewrite to serve such page cache files.

9.3. How Phusion Passenger detects whether a virtual host is a web application

After you've read the deployment instructions you might wonder how Phusion Passenger knows that the DocumentRoot points to a web application that Phusion Passenger is able to serve, and how it knows what kind of web application it is (e.g. Rails or Rack).

Phusion Passenger checks whether the virtual host is a Rails application by checking whether the following file exists:

```
dirname(DocumentRoot) + "/config/environment.rb"
```

If you're not a programmer and don't understand the above pseudo-code snippet, it means that Phusion Passenger will:

1. Extract the parent directory filename from the value of the DocumentRoot directory.
2. Append the text "/config/environment.rb" to the result, and check whether the resulting filename exists.

So suppose that your document root is */webapps/foo/public*. Phusion Passenger will check whether the file */webapps/foo/config/environment.rb* exists.

Note that Phusion Passenger does **not** resolve any symlinks in the document root path by default since version 2.2.0 — in contrast to versions earlier than 2.2.0, which do resolve symlinks. So for example, suppose that your DocumentRoot points to */home/www/example.com*, which in turn is a symlink to */webapps/example.com/public*. In versions earlier than 2.2.0, Phusion Passenger will check whether */webapps/example.com/config*

/environment.rb exists because it resolves all symlinks. Phusion Passenger 2.2.0 and later however will check for */home/www/config/environment.rb*. This file of course doesn't exist, and as a result Phusion Passenger will not activate itself for this virtual host, and you'll most likely see an Apache mod_dirindex directory listing.

If you need the old symlink-resolving behavior for whatever reason, then you can turn on PassengerResolveSymlinksInDocumentRoot.

Another way to solve this situation is to explicitly tell Phusion Passenger what the correct application root is through the PassengerAppRoot configuration directive.

Autodetection of Rack applications happens through the same mechanism, exception that Phusion Passenger will look for *config.ru* instead of *config/environment.rb*.

10. Appendix A: About this document

Phusion Passenger is brought to you by Phusion.

11. Appendix B: Terminology

11.1. Application root

The root directory of an application that's served by Phusion Passenger.

In case of Ruby on Rails applications, this is the directory that contains *Rakefile*, *app/*, *config/*, *public/*, etc. In other words, the directory pointed to by `RAILS_ROOT`. For example, take the following directory structure:

```
/apps/foo/       <------ This is the Rails application's application root!
   |
   +- app/
   |   |
   |   +- controllers/
   |   |
   |   +- models/
   |   |
   |   +- views/
   |
   +- config/
   |   |
   |   +- environment.rb
   |   |
   |   +- ...
   |
```

```
   +- public/
   |   |
   |   +- ...
   |
   +- ...
```

In case of Rack applications, this is the directory that contains *config.ru*. For example, take the following directory structure:

```
/apps/bar/      <----- This is the Rack application's application root!
   |
   +- public/
   |    |
   |    +- ...
   |
   +- config.ru
   |
   +- ...
```

In case of Python (WSGI) applications, this is the directory that contains *passenger_wsgi.py*. For example, take the following directory structure:

```
/apps/baz/      <----- This is the WSGI application's application root!
   |
   +- public/
   |    |
   |    +- ...
   |
   +- passenger_wsgi.py
   |
   +- ...
```

12. Appendix C: Spawning methods explained

At its core, Phusion Passenger is an HTTP proxy and process manager. It spawns Ruby on Rails/Rack/WSGI worker processes (which may also be referred to as *backend processes*), and forwards incoming HTTP request to one of the worker processes.

While this may sound simple, there's not just one way to spawn worker processes. Let's go over the different spawning methods. For simplicity's sake, let's assume that we're only talking about Ruby on Rails applications.

12.1. The most straightforward and traditional way: conservative spawning

Phusion Passenger could create a new Ruby process, which will then load the Rails application along with the entire Rails framework. This process will then enter an request handling main loop.

This is the most straightforward way to spawn worker processes. If you're familiar with the Mongrel application server, then this approach is exactly what mongrel_cluster performs: it creates N worker processes, each which loads a full copy of the Rails application and the Rails framework in memory. The Thin application server employs pretty much the same approach.

Note that Phusion Passenger's version of conservative spawning differs slightly from

mongrel_cluster. Mongrel_cluster creates entirely new Ruby processes. In programmers jargon, mongrel_cluster creates new Ruby processes by forking the current process and exec()-ing a new Ruby interpreter. Phusion Passenger on the other hand creates processes that reuse the already loaded Ruby interpreter. In programmers jargon, Phusion Passenger calls fork(), but not exec().

12.2. The smart spawning method

Smart spawning is supported for all Ruby applications but not for WSGI applications.

While conservative spawning works well, it's not as efficient as it could be because each worker process has its own private copy of the Rails application as well as the Rails framework. This wastes memory as well as startup time.

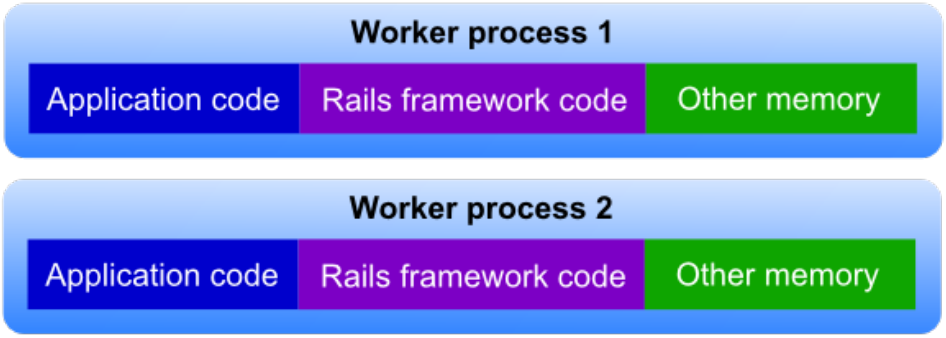


Figure: Worker processes and conservative spawning. Each worker process has its own private copy of the application code and Rails framework code.

It is possible to make the different worker processes share the memory occupied by application and Rails framework code, by utilizing so-called copy-on-write semantics of the virtual memory system on modern operating systems. As a side effect, the startup time is also reduced. This is technique is exploited by Phusion Passenger's *smart* and *smart-lv2* spawn methods.

12.2.1. How it works

When the *smart-lv2* spawn method is being used, Phusion Passenger will first create a so-called *ApplicationSpawner server* process. This process loads the entire Rails application along with the Rails framework, by loading *environment.rb*. Then, whenever Phusion Passenger needs a new worker process, it will instruct the ApplicationSpawner server to do so. The ApplicationSpawner server will create a worker new process that reuses the already loaded Rails application/framework. Creating a worker process through an already running ApplicationSpawner server is very fast, about 10 times faster than loading the Rails application/framework from scratch. If the Ruby interpreter is copy-on-write friendly (that is, if you're running Ruby Enterprise Edition) then all created worker processes will share as much common memory as possible. That is, they will all share the same application and Rails framework code.

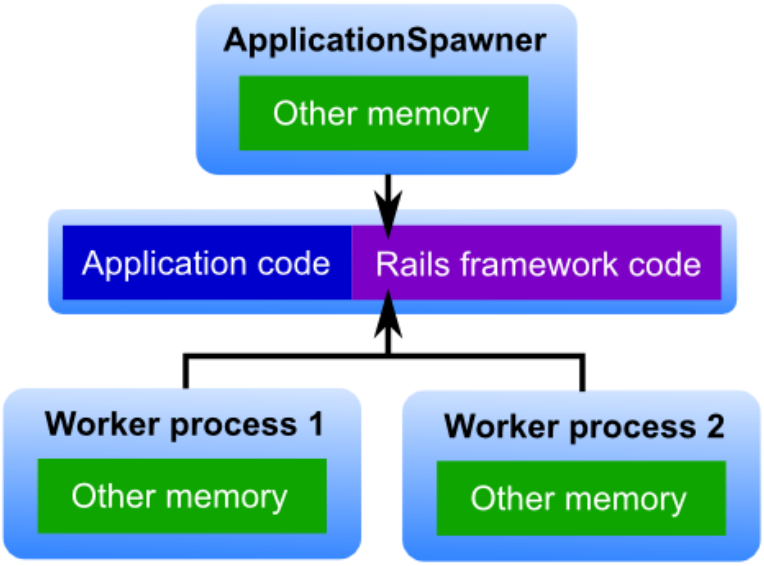


Figure: Worker processes and the smart-lv2 spawn method. All worker processes, as well as the ApplicationSpawner, share the same application code and Rails framework code.

The *smart* spawn method goes even further, by caching the Rails framework in another process called the *FrameworkSpawner server*. This process only loads the Rails framework, not the application. When a FrameworkSpawner server is instructed to create a new worker process, it will create a new ApplicationSpawner to which the instruction will be delegated. All those ApplicationSpawner servers, as well as all worker processes created by those ApplicationSpawner servers, will share the same Rails framework code.

The *smart-lv2* method allows different worker processes that belong to the same application to share memory. The *smart* method allows different worker processes - that happen to use the same Rails version - to share memory, even if they don't belong to the same application.

Notes:

- Vendored Rails frameworks cannot be shared by different applications, even if both vendored Rails frameworks are the same version. So for efficiency reasons we don't recommend vendoring Rails.
- ApplicationSpawner and FrameworkSpawner servers have an idle timeout just like worker processes. If an ApplicationSpawner/FrameworkSpawner server hasn't been instructed to do anything for a while, it will be shutdown in order to conserve memory. This idle timeout is configurable.

12.2.2. Summary of benefits

Suppose that Phusion Passenger needs a new worker process for an application that uses Rails 2.2.1.

- If the *smart-lv2* spawning method is used, and an ApplicationSpawner server for this application is already running, then worker process creation time is about 10 times faster than conservative spawning. This worker process will also share application and Rails framework code memory with the ApplicationSpawner server and the worker processes that had been spawned by this ApplicationSpawner server.
- If the *smart* spawning method is used, and a FrameworkSpawner server for Rails 2.2.1 is already running, but no ApplicationSpawner server for this application is running, then worker process creation time is about 2 times faster than conservative spawning. If there is an ApplicationSpawner server for this application running, then worker process creation time is about 10 times faster. This worker process will also share application and Rails framework code memory with the ApplicationSpawner and FrameworkSpawner servers.

You could compare ApplicationSpawner and FrameworkSpawner servers with stem cells, that have the ability to quickly change into more specific cells (worker process).

In practice, the smart spawning methods could mean a memory saving of about 33%, assuming that your Ruby interpreter is copy-on-write friendly.

Of course, smart spawning is not without gotchas. But if you understand the gotchas you can easily reap the benefits of smart spawning.

12.3. Smart spawning gotcha #1: unintential file descriptor sharing

Because worker processes are created by forking from an ApplicationSpawner server, it will share all file descriptors that are opened by the ApplicationSpawner server. (This is part of the semantics of the Unix *fork()* system call. You might want to Google it if you're not familiar with it.) A file descriptor is a handle which can be an opened file, an opened socket connection, a pipe, etc. If different worker processes write to such a file descriptor at the same time, then their write calls will be interleaved, which may potentially cause problems.

The problem commonly involves socket connections that are unintentially being shared. You can fix it by closing and reestablishing the connection when Phusion Passenger is creating a new worker process. Phusion Passenger provides the API call `PhusionPassenger.on_event(:starting_worker_process)` to do so. So you could insert the following code in your *environment.rb*:

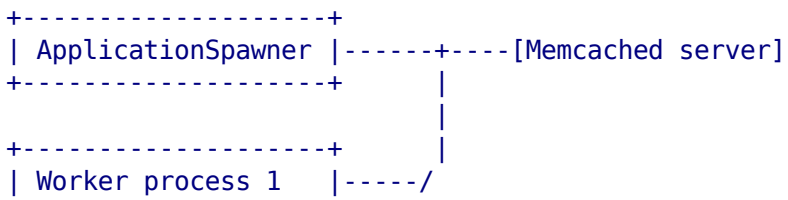
```
if defined?(PhusionPassenger)
    PhusionPassenger.on_event(:starting_worker_process) do |forked|
        if forked
            # We're in smart spawning mode.
            ... code to reestablish socket connections here ...
        else
            # We're in conservative spawning mode. We don't need to do anything.
        end
    end
end
```

Note that Phusion Passenger automatically reestablishes the connection to the database upon creating a new worker process, which is why you normally do not encounter any database issues when using smart spawning mode.

12.3.1. Example 1: Memcached connection sharing (harmful)

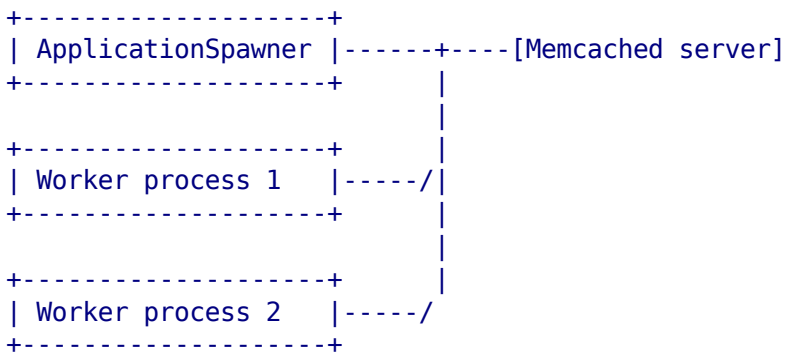
Suppose we have a Rails application that connects to a Memcached server in *environment.rb*. This causes the ApplicationSpawner to have a socket connection (file descriptor) to the Memcached server, as shown in the following figure:

Phusion Passenger then proceeds with creating a new Rails worker process, which is to process incoming HTTP requests. The result will look like this:



Since a *fork()* makes a (virtual) complete copy of a process, all its file descriptors will be copied as well. What we see here is that ApplicationSpawner and Worker process 1 both share the same connection to Memcached.

Now supposed that your site gets Slashdotted and Phusion Passenger needs to spawn another worker process. It does so by forking ApplicationSpawner. The result is now as follows:

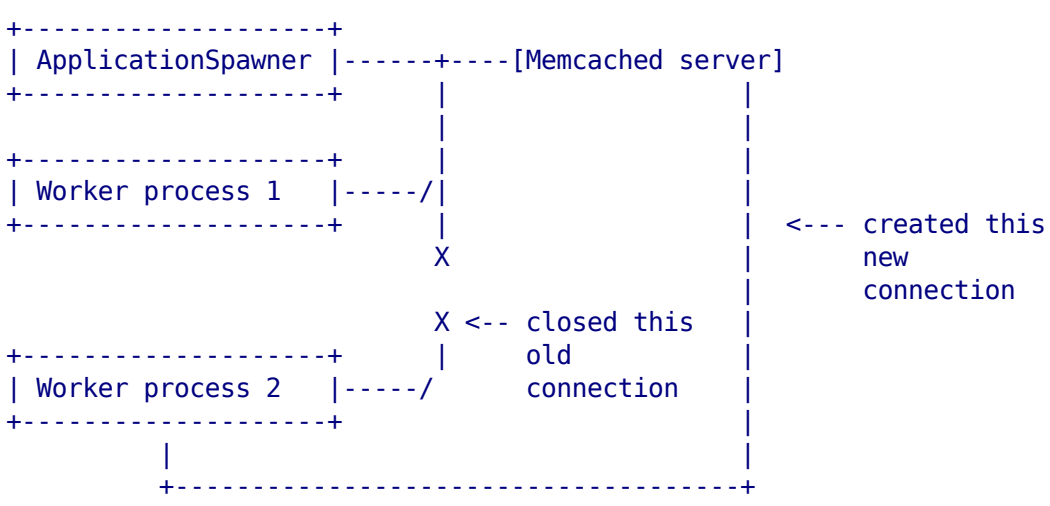


As you can see, Worker process 1 and Worker process 2 have the same Memcache connection.

Suppose that users Joe and Jane visit your website at the same time. Joe's request is handled by Worker process 1, and Jane's request is handled by Worker process 2. Both worker processes want to fetch something from Memcached. Suppose that in order to do that, both handlers need to send a "FETCH" command to Memcached.

But suppose that, after worker process 1 having only sent "FE", a context switch occurs, and worker process 2 starts sending a "FETCH" command to Memcached as well. If worker process 2 succeeds in sending only one bye, *F*, then Memcached will receive a command which begins with "FEF", a command that it does not recognize. In other words: the data from both handlers get interleaved. And thus Memcached is forced to handle this as an error.

This problem can be solved by reestablishing the connection to Memcached after forking:



Worker process 2 now has its own, separate communication channel with Memcached. The code in *environment.rb* looks like this:

```
if defined?(PhusionPassenger)
    PhusionPassenger.on_event(:starting_worker_process) do |forked|
        if forked
            # We're in smart spawning mode.
```

```
            reestablish_connection_to_memcached
        else
            # We're in conservative spawning mode. We don't need to do anything.
        end
    end
end
```

12.3.2. Example 2: Log file sharing (not harmful)

There are also cases in which unintential file descriptor sharing is not harmful. One such case is log file file descriptor sharing. Even if two processes write to the log file at the same time, the worst thing that can happen is that the data in the log file is interleaved.

To guarantee that the data written to the log file is never interleaved, you must synchronize write access via an inter-process synchronization mechanism, such as file locks. Reopening the log file, like you would have done in the Memcached example, doesn't help.

12.4. Smart spawning gotcha #2: the need to revive threads

Another part of the *fork()* system call's semantics is the fact that threads disappear after a fork call. So if you've created any threads in environment.rb, then those threads will no longer be running in newly created worker process. You need to revive them when a new worker process is created. Use the `:starting_worker_process` event that Phusion Passenger provides, like this:

```
if defined?(PhusionPassenger)
    PhusionPassenger.on_event(:starting_worker_process) do |forked|
        if forked
            # We're in smart spawning mode.
            ... code to revive threads here ...
        else
            # We're in conservative spawning mode. We don't need to do anything.
        end
    end
end
```

12.5. Smart spawning gotcha #3: code load order

This gotcha is only applicable to the *smart* spawn method, not the *smart-lv2* spawn method.

If your application expects the Rails framework to be not loaded during the beginning of *environment.rb*, then it can cause problems when an ApplicationSpawner is created from a FrameworkSpawner, which already has the Rails framework loaded. The most common case is when applications try to patch Rails by dropping a modified file that has the same name as Rails's own file, in a path that comes earlier in the Ruby search path.

For example, suppose that we have an application which has a patched version of *active_record/base.rb* located in *RAILS_ROOT/lib/patches*, and *RAILS_ROOT/lib/patches* comes first in the Ruby load path. When conservative spawning is used, the patched version of *base.rb* is properly loaded. When *smart* (not *smart-lv2*) spawning is used, the original *base.rb* is used because it was already loaded, so a subsequent `require "active_record/base"` has no effect.

Last updated 2010-10-10 20:52:03 CEST