Sinatra Recipes

Community contributed recipes and techniques

Fork me on GitHub

Select a topic

Deployment ¶ ↑

This section covers techniques and software used for deploying your Sinatra applications. Whether you're deploying to Heroku or a more advanced setup with nginx and unicorn, you should be able to find what your looking.

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Nginx Proxied to Unicorn ¶ ↑

Nginx and Unicorn combine to provide a very powerful setup for deploying your Sinatra applications. This guide will show you how to effectively setup this combination for deployment.

Installation ¶ ↑

First thing you will need to do is get nginx installed on your system. This should be handled by your operating systems package manager.

For more information on installing nginx, check the official docs.

Once you have nginx installed, you can install unicorn with rubygems:

```
gem install unicorn
```

Now that's done we can setup a basic Rack application in Sinatra.

The Example Application ¶ ↑

To start our example application, let's first create a `config.ru` in our application root.

```
require "rubygems"
require "sinatra"

require File.expand_path '../myapp.rb', __FILE__

run MyApp
```

Let's now use the `myapp.rb` that we specified in our Rack config file as our Sinatra application.

```
require "rubygems"
require "sinatra/base"

class MyApp < Sinatra::Base

  get '/' do
    'Hello, nginx and unicorn!'
  end

end
```

Now that we have our application in place, let's get on to configuring our proxy.

Configuration ¶↑

So if you've made it this far you should have a Sinatra application ready with nginx and unicorn installed. You're ready to move on to configuring the web server.

Unicorn

Configuring unicorn is really easy and provides an easy to use Ruby DSL for doing so. In our application's root we'll first need to make a couple directories, if you haven't already:

```
mkdir tmp
mkdir tmp/sockets
mkdir tmp/pids
mkdir log
```

Once those are in place, we're ready to setup our `unicorn.rb` configuration.

```
# set path to app that will be used to configure unicorn,
# note the trailing slash in this example
@dir = "/path/to/app/"

worker_processes 2
working_directory @dir

timeout 30

# Specify path to socket unicorn listens to,
# we will use this in our nginx.conf later
```

```
listen "#{@dir}tmp/sockets/unicorn.sock", :backlog => 64

# Set process id path
pid "#{@dir}tmp/pids/unicorn.pid"

# Set log file paths
stderr_path "#{@dir}log/unicorn.stderr.log"
stdout_path "#{@dir}log/unicorn.stdout.log"
```

As you can see, unicorn is extremely simple to setup. Let's move onto nginx now, soon enough you'll be well on your way to serving up all kinds of great Rack applications!

nginx

Nginx is a little more difficult to configure than unicorn, but still a fairly straightforward process.

In this example we'll be putting all of our configuration in the `nginx.conf` file of our nginx installation. You could alternatively separate some of the configuration out into `sites-enabled` and other nginx conventions. However, for most common and simple implementations this guide should do the trick.

```
# this sets the user nginx will run as,
#and the number of worker processes
user nobody nogroup;
worker_processes  1;

# setup where nginx will log errors to
# and where the nginx process id resides
error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;

events {
  worker_connections  1024;
  # set to on if you have more than 1 worker_processes
  accept_mutex off;
}

http {
  include       /etc/nginx/mime.types;

  default_type application/octet-stream;
  access_log /tmp/nginx.access.log combined;

  # use the kernel sendfile
  sendfile        on;
  # prepend http headers before sendfile()
  tcp_nopush     on;

  keepalive_timeout  5;
  tcp_nodelay        on;

  gzip  on;
  gzip_vary on;
```

```
    gzip_min_length 500;

    gzip_disable "MSIE [1-6]\.(?!.*SV1)";
    gzip_types text/plain text/xml text/css
       text/comma-separated-values
       text/javascript application/x-javascript
       application/atom+xml image/x-icon;

    # use the socket we configured in our unicorn.rb
    upstream unicorn_server {
      server unix:/path/to/app/tmp/sockets/unicorn.sock
          fail_timeout=0;
    }

    # configure the virtual host
    server {
      # replace with your domain name
      server_name my-sinatra-app.com;
      # replace this with your static Sinatra app files, root + public
      root /path/to/app/public;
      # port to listen for requests on
      listen 80;
      # maximum accepted body size of client request
      client_max_body_size 4G;
      # the server will close connections after this time
      keepalive_timeout 5;

      location / {
        try_files $uri @app;
      }

      location @app {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        # pass to the upstream unicorn server mentioned above
        proxy_pass http://unicorn_server;
      }
    }
  }
```

Once you replace `path/to/app` with the location to your Sinatra application, you should be able now start your application and web server.

Starting the server

First thing you will need to do is boot up the unicorn processes:

```
  unicorn -c path/to/unicorn.rb -E development -D
```

It's important to note the flags here, `-c` is path to your unicorn configuration. `-E` is the Rack environment for your application to run under and `-D` will daemonize the process.

Lastly, let's start up nginx. On most debian-based systems you can use the following command:

```
/etc/init.d/nginx start
```

However, you should check with your distribution as to where the nginx daemon resides.

Now you should have successfully deployed your Sinatra application on nginx and unicorn.

Stopping the server

So now that you're using nginx and unicorn, at some point you might end up asking yourself: How do I stop this thing?

Remember that pids folder we created earlier? Well in there is the pid for the `master` unicorn process, so let's try to use that first:

```
cat /path/to/app/tmp/pids/unicorn.pid | xargs kill -QUIT
```

What we're doing is getting the PID from the pidfile created with Unicorn started and then asking the OS to stop the process. We use the QUIT signal which lets Unicorn shutdown gracefully, but waiting for its workers to finish.

If that doesn't work though, you might want to try:

```
$ ps -ax | grep unicorn
```

This will output the processes running unicorn, in the first column should be the process id (pid). In order to stop unicorn in it's tracks:

```
kill -9 <PID>
```

There should be a `master` process which once that is killed, the workers should follow. Feel free to search the processes again to make sure they've all stopped before restarting.

`kill -9` sends an INT signal to the Unicrons and guarantees (usually) that they'll be stopped, but it means that Unicorn might not clean up after itself properly so you should have a check and run the following just to be sure:

```
rm /path/to/app/tmp/sockets/unicorn.socket
rm /path/to/app/tmp/pids/unicorn.pid
```

To stop nginx you can use a similar technique as above, or if you've got the nginx init scripts installed on any debian-based system use:

```
sudo /etc/init.d/nginx stop
```

That should wrap things up for deploying nginx and unicorn, for more information on stopping the server look into `man ps` and `man kill`.

Resources ¶↑

- unicorn source on GitHub
- original unicorn announcement
- official unicorn homepage
- unicorn rdoc
- nginx official homepage
- nginx wiki

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Apache and Passenger (mod rails) ¶ ↑

Hate deployment via FastCGI? You're not alone. But guess what, Passenger supports Rack; and this book tells you how to get it all going.

You can find additional documentation at the official modrails website.

The easiest way to get started with Passenger is via the gem.

Installation ¶ ↑

First you will need to have Apache installed, then you can move onto installing passenger and the apache passenger module.

You have a number of options when installing phusion passenger, however the gem is likely the easiest way to get started.

```
gem install passenger
```

Once you've got that installed you can build the passenger apache module.

```
passenger-install-apache2-module
```

Follow the instructions given by the installer.

Deploying your app

Passenger lets you easily deploy Sinatra apps through the Rack interface.

There are some assumptions made about your application, however, particularly the `tmp` and `public` sub-directories of your application.

In order to fit these prerequisites, simply make sure you have the following setup:

```
mkdir public
mkdir tmp
config.ru
```

The public directory is for serving static files and tmp directory is for the `restart.txt` application restart mechanism. `config.ru` is where you will place your rackup configuration.

Rackup

Once you have these directories in place, you can setup your applications rackup file, `config.ru`.

```
require 'rubygems'
require 'sinatra'
require File.expand_path '../app.rb', __FILE__

run Sinatra::Application
```

Virtual Host

Next thing you'll have to do is setup the Apache Virtual Host for your app.

```
<VirtualHost *:80>
    ServerName www.yourapplication.com
    DocumentRoot /path/to/app/public
    <Directory /path/to/app/public>
        Allow from all
        Options -MultiViews
    </Directory>
</VirtualHost>
```

That should just about do it for your basic apache and passenger configuration. For more specific information please visit the official modrails documentation.

A note about restarting the server

Once you've got everything configured it's time to restart Apache.

On most debian-based systems you should be able to:

```
sudo apache2ctl stop
# then
sudo apache2ctl start
```

To restart Apache. Check the link above for more detailed information.

In order to restart the Passenger application, all you need to do is run this simple command for your application root:

```
touch tmp/restart.txt
```

You should be up and running now with Phusion Passenger and Apache, if you run into any problems please consult the official docs.

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Dreamhost Deployment via Passenger

You can deploy your Sinatra apps to Dreamhost, a shared web hosting service, via Passenger with relative ease. Here's how.

Setting up the account in the Dreamhost interface

```
Domains -> Manage Domains -> Edit (web hosting column)
Enable 'Ruby on Rails Passenger (mod_rails)'
Add the public directory to the web directory box. So if you were using 'rails.com', it would
change to 'rails.com/public'
Save your changes
```

Creating the directory structure

```
domain.com/
domain.com/tmp
domain.com/public
# a vendored version of sinatra - not necessary if you use the gem
domain.com/sinatra
```

Rack integration

Here is an example config.ru file that does two things. First, it requires your main app file, whatever it's called. In the example, it will look for `myapp.rb`. Second, run your application. If you're subclassing, use the subclass's name, otherwise use Sinatra::Application.

```
require File.expand_path '../myapp.rb', __FILE__

run Sinatra::Application
```

A very simple Sinatra application

```
# this is myapp.rb referred to above
require 'sinatra'
get '/' do
  "Worked on dreamhost"
end

get '/foo/:bar' do
  "You asked for foo/#{params[:bar]}"
end
```

And that's all there is to it! Once it's all setup, point your browser at your domain, and you should see a 'Worked on Dreamhost' page. To restart the application after making changes, you need to run `touch tmp/restart.txt`.

Please note that currently passenger 2.0.3 has a bug where it can cause Sinatra to not find the view directory. In that case, add `:views => '/path/to/views/'` to the Sinatra options in your Rackup file.

You may encounter the dreaded "Ruby (Rack) application could not be started" error with this message "can't activate rack (>= 0.9.1, < 1.0, runtime), already activated rack-0.4.0". This happens because DreamHost has version 0.4.0 installed, when recent versions of Sinatra require more recent versions of Rack. The solution is to explicitly require the rack and sinatra gems in your config.ru. Add the following two lines to the start of your config.ru file:

```
require '/home/USERNAME/.gem/ruby/1.8/gems/rack-VERSION-OF-RACK-GEM-YOU-HAVE-
INSTALLED/lib/rack.rb'
require '/home/USERNAME/.gem/ruby/1.8/gems/sinatra-VERSION-OF-SINATRA-GEM-YOU-HAVE-
INSTALLED/lib/sinatra.rb'
```

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project

after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

FastCGI ¶ ↑

The standard method for deployment is to use Thin or Mongrel, and have a reverse proxy (lighttpd, nginx, or even Apache) point to your bundle of servers.

But that isn't always possible. Cheaper shared hosting (like Dreamhost) won't let you run Thin or Mongrel, or setup reverse proxies (at least on the default shared plan).

Luckily, Rack supports various connectors, including CGI and FastCGI. Unluckily for us, FastCGI doesn't quite work with the current Sinatra release without some tweaking.

Deployment with Sinatra version 0.9 ¶ ↑

From version 0.9.0 Sinatra requires Rack 0.9.1, however FastCGI wrapper from this version seems not working well with Sinatra unless you define your application as a subclass of Sinatra::Application class and run this application directly as a Rack application.

Steps to deploy via FastCGI:

- htaccess
- subclass your application as Sinatra::Application
- dispatch.fcgi

.htaccess ¶ ↑

```
RewriteEngine on

AddHandler fastcgi-script .fcgi
Options +FollowSymLinks +ExecCGI

RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
```

Subclass your application as Sinatra::Application

```ruby
# my_sinatra_app.rb
class MySinatraApp < Sinatra::Application
  # your sinatra application definitions
end
```

dispatch.fcgi - Run this application directly as a Rack application

```ruby
#!/usr/local/bin/ruby

require 'rubygems'
require 'rack'

fastcgi_log = File.open("fastcgi.log", "a")
STDOUT.reopen fastcgi_log
STDERR.reopen fastcgi_log
STDOUT.sync = true

module Rack
  class Request
    def path_info
      @env["REDIRECT_URL"].to_s
    end
    def path_info=(s)
      @env["REDIRECT_URL"] = s.to_s
    end
  end
end

load 'my\_sinatra\_app.rb'

builder = Rack::Builder.new do
  map '/' do
    run MySinatraApp.new
  end
end

Rack::Handler::FastCGI.run(builder)
```

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

JRuby ¶ ↑

The JRuby platform is an excellent deployment choice for just about any Sinatra application. With its seamless integration to the Java ecosystem your application can immediately benefit from a rock-solid VM that leaves most language platforms drooling for more. Years of engineering efforts in scaling and performance, community-driven libraries and frameworks, and of course truly multi-threaded applications (no green threads, no GILs) are just some of those immediate benefits.

The deployment landscape for JRuby is easy to navigate. You can deploy using Mongrel, Thin (experimental), Webrick (but who would do that?), and even Java-centric application containers like Glassfish, Tomcat, or JBoss. We will discuss just a few of these approaches.

The Basics ¶ ↑

There are several ways to install JRuby, but that is beyond the scope of this article. For now, we will assume you are using the excellent RVM , from Wayne Seguin, and proceed from there.

```
rvm install jruby    # (if you haven't already)
rvm use jruby
gem install mongrel
```

Create a classic Sinatra application and save it to hello.rb:

```
require 'rubygems'
require 'sinatra'
```

```
get '/' do
  %Q{
    <html>
      <body>
        Hello from the
        <strong>wonderful</strong>
        world of JRuby!
      </body>
    </html>
  }
end
```

Now, you're ready to fire it up.

```
ruby /path/to/hello.rb
```

That's it. Open localhost:4567 in your browser and your Sinatra application should greet you from the **wonderful** world of JRuby.

Of course, you can use a config.ru and rackup as well:

```
require 'rubygems'
require 'sinatra'
require File.expand_path '../hello.rb', __FILE__
run Sinatra::Application
```

Then launch app with the following command:

```
rackup config.ru
```

Now go to localhost:9292 and you'll see the same **wonderful** greeting.

Deployment with Trinidad

Trinidad is a RubyGem that allows you to run any Rack based application within an embedded Apache Tomcat container.

Again, with our example above, you would do the following:

```
gem install trinidad
trinidad -r config.ru -p 4567 -g ERROR
```

The *-g* option sets the logging level to error.

Deployment with TorqueBox

TorqueBox is built on the JBoss Application Server. Out of the mouths of the TorqueBox developers

themselves:

> JBoss AS includes high-performance clustering, caching and messaging functionality. By building Ruby capabilities on top of this foundation, your Ruby applications gain more capabilities right out-of-the-box.

Deployment on TorqueBox is pretty simple, just like the other deployments we've seen so far -- it embraces Rake for accomplishing its tasks. For installation help checkout their tutorial.

During the installation of TorqueBox you probably set a JBOSS_HOME variable, if not make sure you have set it like this:

```
export JBOSS_HOME=/path/to/TorqueBox/jboss
```

In your Rakefile add this line:

```
require 'TorqueBox/tasks'
```

Now, you are ready to deploy your application:

```
rake TorqueBox:deploy
```

To deploy a production version of your application:

```
RACK_ENV=production rake TorqueBox:deploy
```

To take your application down, you run:

```
rake TorqueBox:undeploy
```

TorqueBox also allows you deploy to a non-root context:

```
rake TorqueBox:deploy['/hello']
```

Conclusion ¶ ↑

That's it. Pretty simple, eh?! There are other players in the field, like Glassfish and Jetty.

Becoming familiar with each of these deployment options, what one offers over the other, strengths and weaknesses, etc. is not an easy task. So here are some basic tips:

- If your application just needs a rock-solid, easy deployment without all of the enterprise-y bells and whistles that many Java app servers offer you, choose a simple bare-bones JRuby deployment with Mongrel or Jetty with jetty-rackup.
- If you need highly scalable, clustered deployments consider TorqueBox or Glassfish. Both are solid performers with messaging capabilities built in. TorqueBox may be a simpler approach.
- If your company or service provider already supports Tomcat, use Trinidad.

If you aren't sure... try all of them. I would suggest spending some effort in writing a benchmark suite so that you have a good comparison of what really matters in the end -- performance, memory consumption, disk space, scalability, etc. You could even start with TorqueBox's speedmetal benchmarking suite and craft it to suit your fancy. Read up on TorqueBox's benchmarking efforts at

TorqueBox.org/news/2011/02/23/benchmarking-TorqueBox.

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Lighttpd Proxied to Thin

This will cover how to deploy Sinatra to a load balanced reverse proxy setup using Lighttpd and Thin.

Install Lighttpd and Thin

```
# Figure out lighttpd yourself, it should be handled by your
# linux distro's package manager

# For thin:
gem install thin
```

Create your rackup file

The `require 'app'` line should require the actual Sinatra app you have written.

```
## This is not needed for Thin > 1.0.0
ENV['RACK_ENV'] = "production"

require File.expand_path '../app.rb', __FILE__
```

```
run Sinatra::Application
```

Setup a config.yml

Change the /path/to/my/app path to reflect reality.

```
---
 environment: production
 chdir: /path/to/my/app
 address: 127.0.0.1
 user: root
 group: root
 port: 4567
 pid: /path/to/my/app/thin.pid
 rackup: /path/to/my/app/config.ru
 log: /path/to/my/app/thin.log
 max_conns: 1024
 timeout: 30
 max_persistent_conns: 512
 daemonize: true
```

Setup lighttpd.conf

Change mydomain to reflect reality. Also make sure the first port here matches up with the port setting in config.yml.

```
$HTTP["host"] =~ "(www\.)?mydomain\.com"  {
 proxy.balance = "fair"
 proxy.server =  ("/" =>
                   (
                     ( "host" => "127.0.0.1", "port" => 4567 ),
                     ( "host" => "127.0.0.1", "port" => 4568 )
                   )
                 )
}
```

Start thin and your application.

I have a rake script so I can just call “rake start” rather than typing this in.

```
thin -s 2 -C config.yml -R config.ru start
```

You’re done! Go to mydomain.com/ and see the result! Everything should be setup now, check it out at the domain you setup in your lighttpd.conf file.

Variation – nginx via proxy – The same approach to proxying can be applied to the nginx web server

```
upstream www_mydomain_com {
  server 127.0.0.1:5000;
  server 127.0.0.1:5001;
}

server {
  listen    www.mydomain.com:80
  server_name  www.mydomain.com live;
  access_log /path/to/logfile.log

  location / {
    proxy_pass http://www_mydomain_com;
  }

}
```

Variation – More Thin instances – To add more thin instances, change the `-s 2` parameter on the thin start command to be how ever many servers you want. Then be sure lighttpd proxies to all of them by adding more lines to the proxy statements. Then restart lighttpd and everything should come up as expected.

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Nginx Proxied to Unicorn ¶ ↑

Nginx and Unicorn combine to provide a very powerful setup for deploying your Sinatra applications. This guide will show you how to effectively setup this combination for deployment.

Installation ¶ ↑

First thing you will need to do is get nginx installed on your system. This should be handled by your operating systems package manager.

For more information on installing nginx, check the official docs.

Once you have nginx installed, you can install unicorn with rubygems:

```
gem install unicorn
```

Now that's done we can setup a basic Rack application in Sinatra.

The Example Application ¶ ↑

To start our example application, let's first create a `config.ru` in our application root.

```
require "rubygems"
require "sinatra"

require File.expand_path '../myapp.rb', __FILE__

run MyApp
```

Let's now use the `myapp.rb` that we specified in our Rack config file as our Sinatra application.

```
require "rubygems"
require "sinatra/base"

class MyApp < Sinatra::Base

  get '/' do
    'Hello, nginx and unicorn!'
  end

end
```

Now that we have our application in place, let's get on to configuring our proxy.

Configuration ¶↑

So if you've made it this far you should have a Sinatra application ready with nginx and unicorn installed. You're ready to move on to configuring the web server.

Unicorn

Configuring unicorn is really easy and provides an easy to use Ruby DSL for doing so. In our application's root we'll first need to make a couple directories, if you haven't already:

```
mkdir tmp
mkdir tmp/sockets
mkdir tmp/pids
mkdir log
```

Once those are in place, we're ready to setup our `unicorn.rb` configuration.

```
# set path to app that will be used to configure unicorn,
# note the trailing slash in this example
@dir = "/path/to/app/"

worker_processes 2
working_directory @dir

timeout 30

# Specify path to socket unicorn listens to,
# we will use this in our nginx.conf later
```

```
listen "#{@dir}tmp/sockets/unicorn.sock", :backlog => 64

# Set process id path
pid "#{@dir}tmp/pids/unicorn.pid"

# Set log file paths
stderr_path "#{@dir}log/unicorn.stderr.log"
stdout_path "#{@dir}log/unicorn.stdout.log"
```

As you can see, unicorn is extremely simple to setup. Let's move onto nginx now, soon enough you'll be well on your way to serving up all kinds of great Rack applications!

nginx

Nginx is a little more difficult to configure than unicorn, but still a fairly straightforward process.

In this example we'll be putting all of our configuration in the `nginx.conf` file of our nginx installation. You could alternatively separate some of the configuration out into `sites-enabled` and other nginx conventions. However, for most common and simple implementations this guide should do the trick.

```
# this sets the user nginx will run as,
#and the number of worker processes
user nobody nogroup;
worker_processes  1;

# setup where nginx will log errors to
# and where the nginx process id resides
error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;

events {
  worker_connections  1024;
  # set to on if you have more than 1 worker_processes
  accept_mutex off;
}

http {
  include       /etc/nginx/mime.types;

  default_type application/octet-stream;
  access_log /tmp/nginx.access.log combined;

  # use the kernel sendfile
  sendfile        on;
  # prepend http headers before sendfile()
  tcp_nopush     on;

  keepalive_timeout  5;
  tcp_nodelay        on;

  gzip  on;
  gzip_vary on;
```

```
    gzip_min_length 500;

    gzip_disable "MSIE [1-6]\.(?!.*SV1)";
    gzip_types text/plain text/xml text/css
       text/comma-separated-values
       text/javascript application/x-javascript
       application/atom+xml image/x-icon;

    # use the socket we configured in our unicorn.rb
    upstream unicorn_server {
      server unix:/path/to/app/tmp/sockets/unicorn.sock
          fail_timeout=0;
    }

    # configure the virtual host
    server {
      # replace with your domain name
      server_name my-sinatra-app.com;
      # replace this with your static Sinatra app files, root + public
      root /path/to/app/public;
      # port to listen for requests on
      listen 80;
      # maximum accepted body size of client request
      client_max_body_size 4G;
      # the server will close connections after this time
      keepalive_timeout 5;

      location / {
        try_files $uri @app;
      }

      location @app {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        # pass to the upstream unicorn server mentioned above
        proxy_pass http://unicorn_server;
      }
    }
  }
```

Once you replace `path/to/app` with the location to your Sinatra application, you should be able now start your application and web server.

Starting the server

First thing you will need to do is boot up the unicorn processes:

```
  unicorn -c path/to/unicorn.rb -E development -D
```

It's important to note the flags here, `-c` is path to your unicorn configuration. `-E` is the Rack environment for your application to run under and `-D` will daemonize the process.

Lastly, let's start up nginx. On most debian-based systems you can use the following command:

```
/etc/init.d/nginx start
```

However, you should check with your distribution as to where the nginx daemon resides.

Now you should have successfully deployed your Sinatra application on nginx and unicorn.

Stopping the server

So now that you're using nginx and unicorn, at some point you might end up asking yourself: How do I stop this thing?

Remember that pids folder we created earlier? Well in there is the pid for the `master` unicorn process, so let's try to use that first:

```
cat /path/to/app/tmp/pids/unicorn.pid | xargs kill -QUIT
```

What we're doing is getting the PID from the pidfile created with Unicorn started and then asking the OS to stop the process. We use the QUIT signal which lets Unicorn shutdown gracefully, but waiting for its workers to finish.

If that doesn't work though, you might want to try:

```
$ ps -ax | grep unicorn
```

This will output the processes running unicorn, in the first column should be the process id (pid). In order to stop unicorn in it's tracks:

```
kill -9 <PID>
```

There should be a `master` process which once that is killed, the workers should follow. Feel free to search the processes again to make sure they've all stopped before restarting.

`kill -9` sends an INT signal to the Unicrons and guarantees (usually) that they'll be stopped, but it means that Unicorn might not clean up after itself properly so you should have a check and run the following just to be sure:

```
rm /path/to/app/tmp/sockets/unicorn.socket
rm /path/to/app/tmp/pids/unicorn.pid
```

To stop nginx you can use a similar technique as above, or if you've got the nginx init scripts installed on any debian-based system use:

```
sudo /etc/init.d/nginx stop
```

That should wrap things up for deploying nginx and unicorn, for more information on stopping the server look into `man ps` and `man kill`.

Resources ¶↑

- unicorn source on GitHub
- original unicorn announcement
- official unicorn homepage
- unicorn rdoc
- nginx official homepage
- nginx wiki

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top