Deploying your Rails application to a clean Ubuntu 10.04 install using Nginx and Unicorn

If you're a developer, setting up a server might not look like something you'd do yourself. Most of the time you have a sysadmin in your team/company/business that can do the job for you. But knowing how to configure a server might make your life easier if you're rolling on your own so let's take a look at how we can configure a clean slate Ubuntu 10.04 server machine to run a Ruby on Rails application.

This example used a just created Ubuntu 10.04 image from Rackspace, but you should be able to follow it on any Ubuntu 10.04 install, even a local one.

With this tutorial you'll learn how to:

- Install the libraries usually necessary to run Ruby on Rails application;
- Setup Nginx as the HTTP server proxy and statics assets server;
- Setup Unicorn as the application server that's going to run your application;
- Setup Monit to watch over the processes
- Setup some basic firewall rules
- And finally deploy your Ruby on Rails application to it;

Remember that this tutorial is given in an "as is" basis and **you should** backup all system files we're changing here.

Aliasing the server at your ssh config

Whenever I'm trying to access the server at this tutorial I'll be using the "rs" alias, this is an SSH defined alias that you can do yourself. First, create a ".ssh" folder at your home directory:

```
mkdir ~/.ssh
```

Then, with your favorite text editor, create a file called "config" on it:

```
mate ~/.ssh/config
```

The file content for me is as follows (you should change the IP to match your remote server IP):

```
Host rs
Hostname 184.106.215.175
User root
```

This will tell the SSH program that whenever I try to access a server named as "rs" it should login into the IP "184.106.217.176" and with the user "root" (don't worry, we'll change this later). So, to login to your server I can just type:

```
ssh rs
```

And it should ask for your password.

Generating local SSH keys

You can jump to the next section if you already have SSH Keys for your user, if you don't have or don't know what they are go on reading. The SSH keys we're going to generate here will be used to simplify your login over SSH to the remote server.

With the keys in place you won't need to type in a password to access the server, the remote server will check your keys and and if they match the keys on it you'll be logged in. Here's how to generate a key:

```
ssh-keygen -t dsa
```

This will create a ".ssh" folder under you home directory with two files:

- id_dsa – your **private** key that should not be shared to ANYONE
- id_dsa.pub – your **public** key, the one you will be copying to your servers and services like GitHub

Logging in to the remote server without typing a password

To be able to login to the remote machine without a password, you have to copy your **public** to the remote machine into a user's home directory. The file with the keys should be at "~/.ssh/authorized_keys2".

First, login into the remote server (as root) and create an ".ssh" folder:

```
mkdir ~/.ssh
```

Then, log off and, at your local machine, send your id_dsa.pub file to the remote server with an SCP:

```
scp ~/.ssh/id_dsa.pub root@rs:.ssh/authorized_keys2
```

The "authorized_keys2" file can accept as many keys you'd like, just place each key in a single line.

Installing the required libraries and applications

Logged in as root in the newly created server, you should start by getting the server libraries up to date:

```
apt-get update
apt-get upgrade –y
```

After that, it's time to start installing the Ruby required libraries

```
apt-get install build-essential ruby-full libmagickcore-dev
imagemagick libxml2-dev libxslt1-dev git-core -y
```

With the basic ruby libraries, it's time to select your database. If you're going for MySQL, here's what you would install:

```
apt-get install mysql-server libmysqlclient-dev –y
```

If you're going for PostgreSQL:

```
apt-get install postgresql postgresql-client postgresql-server-dev-
8.4 –y
```

Now we have to install the web server, monit and the firewall:

```
apt-get install nginx monit ufw –y
```

And the last step is to install Rubygems from the source. The Ubuntu provided "gem" install isn't updated as often as the real gem is so it's better to install it from the source. At the time this tutorial was written, Rubygems 1.3.7 was the latest available, you should check at the Rubygems Rubyforge website witch one is the latest version and install it (http://rubyforge.org/projects/rubygems/).

Now let's download Rubygems:

```
wget http://rubyforge.org/frs/download.php/70696/rubygems-1.3.7.tgz
```

Untar it:

```
tar -xzf rubygems-1.3.7.tgz
```

And finally install it:

```
cd rubygems-1.3.7
ruby setup.rb
```

This will install the executable "gem1.8", so we just symlink it to be "gem":

```
ln –s /usr/bin/gem1.8 /usr/bin/gem
```

And then clean up the Rubygems files we downloaded:

```
cd ..
rm –rf rubygems*
```

Before start installing gems, as this is a server environment, you probably don't want to have the **rdoc** and **ri** files generated for every installed gem as this is just going to slow down the gem installation for nothing, as no one will be calling "ri" in this server, to disable this stuff, fire up a text editor in the server:
nano ~/.gemrc

And put this on it:

```
---
:verbose: true
:bulk_threshold: 1000
```

```
install: --no-ri --no-rdoc --env-shebang
:sources:
- http://gemcutter.org
- http://gems.rubyforge.org/
- http://gems.github.com
:benchmark: false
:backtrace: false
update: --no-ri --no-rdoc --env-shebang
:update_sources: true
```

This will avoid ri and rdoc generation while installing gems. And now you can start installing the gems your application will need:

```
gem install rails unicorn will_paginate nokogiri paperclip sunspot
sunspot_rails
```

Add your database gems as needed in this installation too, if you're on MySQL:

```
gem install mysql
```

If you're on PostgreSQL:

```
gem install pg
```

Configuring the default Rails environment

Fire up your editor at the remote server:

```
nano /etc/environment
```

And add the following line to it:

```
export RAILS_ENV=production
```

This will make all "script" calls assume that the environment is "production". The server needs to be rebooted for this change to take effect. This is not needed for Capistrano or Unicorn, as they define the environment themselves.

Setting up a common user

All the work we've been doing so far is as a root user, but our application will not be run as root, but as a common user. Still as root, create a user, any name will do, I usually go for deployer:

```
useradd -m -g staff -s /bin/bash deployer
```

Give this user a password:

```
passwd deployer
```

Now open the "/etc/sudoers" file and make the staff user group be able to perform a "dudo" adding the following line to it:

```
%staff ALL=(ALL) ALL
```

This will allow users with the staff group call the “sudo” command, but will prompt for a password before that.

With the user created, you should login to the machine as “deployer” (or your username) and again create an “.ssh“ folder, after that, send in the public key again:

```
scp ~/.ssh/id_dsa.pub deployer@rs:.ssh/authorized_keys2
```

And also create the same “.gemrc” file you created for the root user at the “deployer” user home directory.

Configuring the Nginx server

Our application is going to be deployed by Capistrano at the “deployer” user home directory, in “/home/deployer/shop”. As we’re going to use Capistrano for deployment, the application will always be available at “/home/deployer/shop/current” and that’s the folder we’re going to use for Nginx.

Given that we’re going to run a single application in this server, we can go on and edit the “/etc/nginx/sites-available/default” file directly, if you’re going to host more than one application in this server you should create separate files for each of them and then symlink them to the “/etc/nginx/sites-enabled/” folder.

Here’s the Nginx configuration file:

```
    # as we're going to use Unicorn as the application server
    # we're not going to use common sockets
    # but Unix sockets for faster communication
    upstream shop {
      # fail_timeout=0 means we always retry an upstream even if it
failed
      # to return a good HTTP response (in case the Unicorn master
nukes a
      # single worker for timing out).

      # for UNIX domain socket setups:
      server unix:/tmp/shop.socket fail_timeout=0;
    }

    server {
        # if you're running multiple servers, instead of "default"
you should
        # put your main domain name here
        listen 80 default;
```

```
        # you could put a list of other domain names this application
answers
        server_name localhost;

        root /home/deployer/shop/current/public;
        access_log  on;
        rewrite_log on;

        location / {
            #all requests are sent to the UNIX socket
            proxy_pass  http://shop;
            proxy_redirect     off;

            proxy_set_header   Host             $host;
            proxy_set_header   X-Real-IP        $remote_addr;
            proxy_set_header   X-Forwarded-For
$proxy_add_x_forwarded_for;

            client_max_body_size       10m;
            client_body_buffer_size    128k;

            proxy_connect_timeout      90;
            proxy_send_timeout         90;
            proxy_read_timeout         90;

            proxy_buffer_size          4k;
            proxy_buffers              4 32k;
            proxy_busy_buffers_size    64k;
            proxy_temp_file_write_size 64k;
        }

        # if the request is for a static resource, nginx should serve
it directly
        # and add a far future expires header to it, making the
browser
        # cache the resource and navigate faster over the website
        location ~ ^/(images|javascripts|stylesheets|system)/  {
          root /home/deployer/shop/current/public;
          expires max;
          break;
        }
```

```
    }
```

The comments in the Nginx file make it self explanatory, but what we're doing is to send all requests that are not for static files directly to the Unicorn backend (using a UNIX socket for that).

Once you have configured the "default" file, you should open the "/etc/nginx/nginx.conf" file as we also have some changes to it. The first one is to change the user that Nginx is going to run as. To avoid permission issues we're going to make Nginx run as "deployer", the user that's going to deploy the application code, this way we can rest safely that no "permission denied" on files will ever happen (at least as long as we don't deploy as another user).

Here's how your Nginx configuration will look like at the end:

```
user deployer staff;
worker_processes  4;

error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
    accept_mutex on;
}

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

    access_log  /var/log/nginx/access.log;

    sendfile on;

    tcp_nopush on; # off may be better for *some* Comet/long-poll
stuff
    tcp_nodelay off; # on may be better for some Comet/long-poll
stuff

    gzip on;
    gzip_http_version 1.0;
    gzip_proxied any;
    gzip_min_length 500;
    gzip_disable "MSIE [1-6]\.";
```

```
    gzip_types text/plain text/html text/xml text/css text/comma-
separated-values text/javascript application/x-javascript
application/atom+xml;


    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

In this file we enable GZIP compression for all requests (and disable it if the client is IE6) and we also define more Nginx workers to do the job, this usually helps while serving static content (if you've got a lot of static content to serve, if you don't you can just go on with just 1 worker). We have also changed Nginx to start as "deployer" with the group "staff", the same user and group of the files he's going to be serving.

And finally, as root, create the following directory:

```
mkdir /usr/local/nginx
```

Now start nginx and see if it's running correctly:

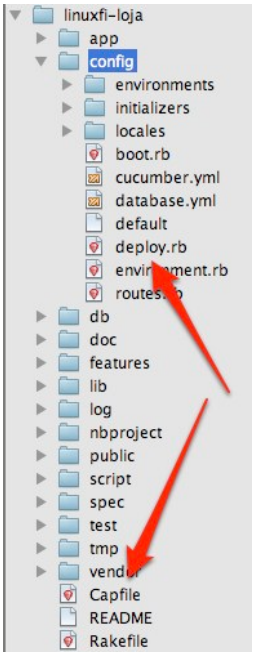
```
/etc/init.d/nginx restart
ps aux | grep nginx
```

Setting up Capistrano

Now it's time to setup Capistrano at your application to perform deployments. In the command line, go to the root folder of your application and (sudo gem install capistrano – if you don't have it already at your local machine) :

```
capify .
```

This will generate a "Capfile" at your project root folder and a "deploy.rb" file at the "config" folder:



The Capfile is just to boot Capistrano, the file we really change is the “deploy.rb”, where we’re going to configure our remote server information and also the deployer user credentials, here’s how the file looks like for the example project:

```
set :use_sudo,            false
#tell git to clone only the latest revision and not the whole repository
set :git_shallow_clone,   1
set :keep_releases,       5
set :application,         "shop"
set :user,                "deployer"
set :password,            "asdfg"
set :deploy_to,           "/home/deployer/shop"
set :runner,              "deployer"
set :repository,          "git@github.com:mauricio/linuxfi-loja.git"
set :scm,                 :git
set :real_revision,  lambda { source.query_revision(revision) { |cmd| capture(cmd) } }
#options necessary to make Ubuntu’s SSH happy
ssh_options[:paranoid]    = false
default_run_options[:pty] = true

role :app, "184.106.215.175"
role :web, "184.106.215.175"
role :db,  "184.106.215.175", :primary => true
```

In this Capistrano file you have to define your server, the user that Capistrano is going to use to login into it (and also the password) and the location of your source code repository.

Each of the roles defined at “deploy.rb” file can have more than one server and you could possibly separate your components, the “app” role is where your application server lives, the “web” role is where the web server lives and the “db” role is where your database is. The “:primary => true” for the :db role tells Capistrano which one is the “master” database if you’re running in a “master-slave” database setup, as if you’re doing this the migrations should only be run in the “master” and not on the slave databases.

Configuring Unicorn

With the Capistrano file defined, let’s create a Unicorn config file for our application, here’s a simple configuration example (this file is placed at the root of our project and named as “unicorn.rb”):

```
# See http://unicorn.bogomips.org/Unicorn/Configurator.html for
complete
# documentation.
worker_processes 4
# Help ensure your application will always spawn in the symlinked
# "current" directory that Capistrano sets up.
working_directory "/home/deployer/shop/current"
# listen on both a Unix domain socket and a TCP port,
# we use a shorter backlog for quicker failover when busy
listen "/tmp/shop.socket", :backlog => 64
# nuke workers after 30 seconds instead of 60 seconds (the default)
timeout 30
# feel free to point this anywhere accessible on the filesystem
user 'deployer', 'staff'
shared_path = “/home/deployer/shop/shared”
pid "#{shared_path}/pids/unicorn.pid"
stderr_path "#{shared_path}/log/unicorn.stderr.log"
stdout_path "#{shared_path}/log/unicorn.stdout.log"
```

You should place the number of worker processes based on the memory you have available in your server and also the number of processor cores it has, leaving too little will make your not use the whole server resources and adding too many of it will surely make everyone wait too much on IO access. So, profile, test and reach your best configuration for your current environment.

Unlike using mongrel or thin, you don’t have to define many processes do be run, Unicorn loads a single process that’s going to take care of loading and serving the requests to all of your workers, so you only have to manage a single process

instead of a process for each worker, something that's going to simplify our work a lot.

Now we also need to configure Unicorn to run as a Daemon (a long running process that's started with Linux itself and not by a user), so it will start itself when the machine boots. To do this we have to create a script that will start/stop/restart our Unicorn server and place it inside the "/etc/init.d" directory, fire up the text editor at the server:

```
nano /etc/init.d/unicorn
```

And here are the contents of the file:

```
#! /bin/sh

### BEGIN INIT INFO
# Provides:          unicorn
# Required-Start:    $local_fs $remote_fs $network $syslog
# Required-Stop:     $local_fs $remote_fs $network $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: starts the unicorn web server
# Description:       starts unicorn
### END INIT INFO

PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
DAEMON=/usr/bin/unicorn_rails
DAEMON_OPTS="-c /home/deployer/shop/current/unicorn.rb -E production
-D"
NAME=unicorn_rails
DESC=unicorn_rails
PID=/home/deployer/shop/shared/pids/unicorn.pid

case "$1" in
  start)
        echo -n "Starting $DESC: "
        $DAEMON $DAEMON_OPTS
        echo "$NAME."
        ;;
  stop)
        echo -n "Stopping $DESC: "
          kill -QUIT `cat $PID`
        echo "$NAME."
        ;;
```

```
  restart)
      echo -n "Restarting $DESC: "
        kill -QUIT `cat $PID`
      sleep 1
      $DAEMON $DAEMON_OPTS
      echo "$NAME."
      ;;
  reload)
        echo -n "Reloading $DESC configuration: "
        kill -HUP `cat $PID`
        echo "$NAME."
        ;;
  *)
      echo "Usage: $NAME {start|stop|restart|reload}" >&2
      exit 1
      ;;
esac

exit 0
```

As you should have noticed by the file, this unicorn configuration is specific for a **single** application, if you have many applications, you'll have to create one file like this one for each of them, which should be something quite easy, as you would only change the values of the PID and DAEMON_OPTS variables.

With the file in place we have to make it executable and tell Linux to load it on startup:

```
chmod +x /etc/init.d/unicorn
update-rc.d unicorn defaults
```

Now Unicorn is ready to boot with your server and run your application.

With the Unicorn stuff ready, commit your changes to the project and push them to your version control system, we're now on to start the deployment phase.

Starting the deployment

Now that almost all the configuration is in place, let's setup the folders in the remote server. At your local machine in the project root folder, run:

```
cap  deploy:setup
```

This will create the whole folder structure for our deployment. After running this, the remote server folder structure should be something like this:

```
root@rails-tutorial:~# ls -l /home/deployer/shop/
```

```
total 8
drwxrwxr-x 2 deployer staff 4096 Aug 25 02:08 releases
drwxrwxr-x 5 deployer staff 4096 Aug 25 02:08 shared
```

Now we have the environment ready for action and we're close to our first deployment. You should now create your database in production and configure the "database.yml" accordingly, as the first deployment is going to run the migrations and needs a pre-existing database to be run.

Once you have the database ready we need to add some code to the deploy.rb file to tell Capistrano how our server is started/stopped. You should add the following lines at the end of your deploy.rb file:

```
namespace :deploy do
  task :start do
    sudo "/etc/init.d/unicorn start"
  end
  task :stop do
    sudo "/etc/init.d/unicorn stop"
  end
  task :restart do
    sudo "/etc/init.d/unicorn reload"
  end
end
```

These are the calls Capistrano is going to make to handle our Unicorn daemon. Look that at the :restart call we do not send a restart to Unicorn but a "reload".

Why this?

When you setup a Unicorn server there are two parts involved, the "master" and the "worker" processes. The master process is the one that handles the workers, starting, stopping and restarting them.

The master process usually doesn't need to be "restarted", only the workers have to so when you update your application you don't want to restart Unicorn, all you want is to restart the workers and this "reload" method does exactly that, it sends a "HUP" signal to the Unicorn master process and it will then start to "stop" the old workers and "start" new ones for their places. The biggest advantage for this behavior is that you will have **zero downtime** as the old workers will be available until the new workers have arrived to serve requests.

The only reason to send a real "restart" to Unicorn is when you're updating the Unicorn gem itself. Before going on to the next step you should check if the machine that contains the source code for your application allows access from your current server, do an "ssh" connection to if to be sure that everything works (if you're using GitHub "ssh git@github.com" and read the section just after this one before running the next command).

If you don't access the source code server at least once your deployment could get stuck at the "add host to known_hosts":

```
The authenticity of host 'github.com (207.97.227.239)' can't be
established.

RSA key fingerprint is
16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'github.com,207.97.227.239' (RSA) to the
list of known hosts.
```

Enough talking, let's go on with the deployment, at your local machine, run:

```
cap deploy:cold
```

The deploy:cold task does the "first" deployment to the machine, setting up new directories, creating log files and everything it needs to start for the first time. After that you should check your server and you should finally see your application running:

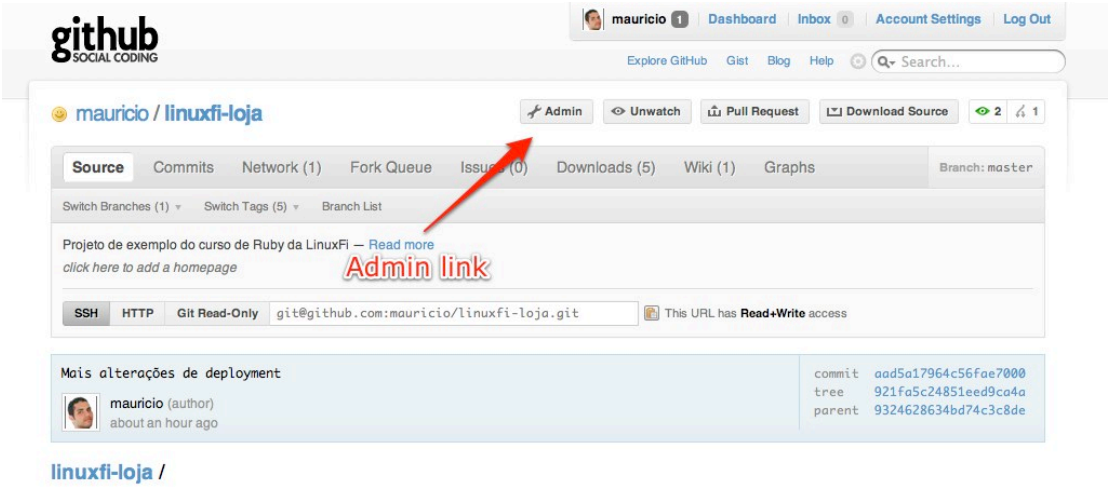
```
root@rails-tutorial:~# ps aux | grep unicorn

root     21567  0.0  0.9  46792  9480 ?        S    03:48   0:00
unicorn_rails master -c /home/deployer/shop/current/unicorn.rb -E
production -D

deployer 21568 15.9  5.8 126272 59788 ?        S    03:48   0:02
unicorn_rails worker[0] -c /home/deployer/shop/current/unicorn.rb -E
production -D

deployer 21569 16.1  5.8 126272 59792 ?        S    03:48   0:02
unicorn_rails worker[1] -c /home/deployer/shop/current/unicorn.rb -E
production -D

deployer 21570 16.0  5.8 126400 59788 ?        S    03:48   0:02
unicorn_rails worker[2] -c /home/deployer/shop/current/unicorn.rb -E
production -D

deployer 21571 16.1  5.8 126268 59776 ?        S    03:48   0:02
unicorn_rails worker[3] -c /home/deployer/shop/current/unicorn.rb -E
production -D
```

Using GitHub as a source code hosting service

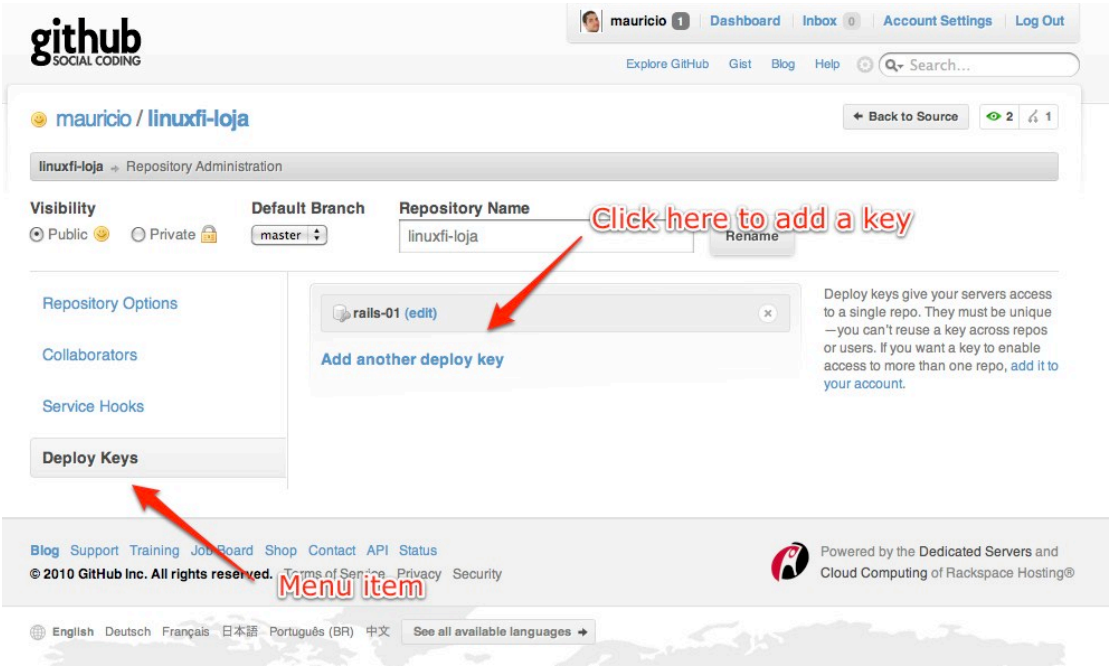
If you're using GitHub with private repos to host your application source code there's another step that needs to be taken, you will have to create a "deploy key" to your server to make it able to access the private GitHub repo. First, login to the server as "deployer" and run:

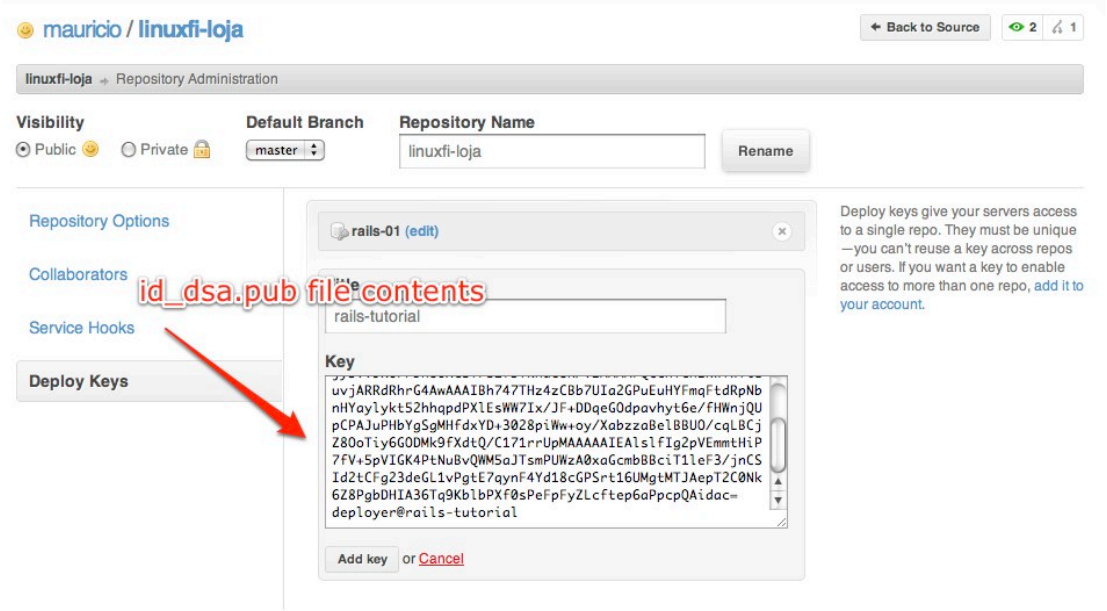
```
ssh-keygen –t dsa
```

Now, get the contents of the "~/.ssh/id_dsa.pub" ("cat ~/.ssh/id_dsa.pub") and at your GitHub repo to go "Admin":



Then go to “Deploy Keys”. You should add the contents of the “id_dsa.pub” file as a deploy key:





Now, your server is ready to pull code from GitHub.

Monitoring your services with Monit

Having Nginx and Unicorn running as daemons that will start if the server reboots is nice, but if something makes Unicorn or Nginx die, we shouldn't need to restart them ourselves, and this is where Monit enters the show.

Monit is a process monitoring solution for Unix environments, where you can define various parameters of process health and let monit look at the processes and see if they're behaving as expected. If they're not doing fine, Monit will try it's configured actions to make things right. In our example, we're going to monitor Nginx and Unicorn, but if you had other processes like Resque or Solr you should also add them as resources watched by Monit.

Open the "/etc/monit/monitrc" at the server:

```
nano /etc/monit/monitrc
```

And add this line right at the end:

```
include /etc/monit/conf.d/*
```

Now we're going to create the Monit configuration files, first the general configuration:

```
nano /etc/monit/conf.d/general.conf
```

And the file content:

```
set httpd port 6874
      allow localhost
      allow moniter:"165907"
```

```
set alert mauricio.linhares@gmail.com
set alert someone-else@gmail.com

set mailserver smtp.gmail.com port 587
        username "some-one@gmail.com" password "123456"
        using tlsv1
        with timeout 30 seconds
set daemon 60
set logfile /var/log/monit.log
```

This configuration tells monit to open a socket connection inside the server to allow commands to be run (if you don't set this configuration, you won't be able to call any commands on it). Then we define to which emails the alerts are going to be sent and finally the email configuration for sending the email alerts.

Now let's tell monit to watch Nginx:

```
nano /etc/monit/conf.d/nginx.conf
```

And the contents:

```
check process unicorn
  with pidfile /var/run/nginx.pid
  start program = "/etc/init.d/nginx start"
  stop program = "/etc/init.d/nginx stop"
  group nginx
```

And finally let's tell monit to watch the Unicorn server:

```
nano /etc/monit/conf.d/unicorn.conf
```

And the contents:

```
check process unicorn
  with pidfile /home/deployer/shop/shared/pids/unicorn.pid
  start program = "/etc/init.d/unicorn start"
  stop program = "/etc/init.d/unicorn stop"
  group unicorn
```

After doing that, open the "/etc/default/monit" and change this:

```
startup=0
```

To this:

```
startup=1
```

Now you can start monit ("/etc/init.d/monit start") and it will start watching over your processes. You can add more tests, like CPU, memory, filesystem usage to your processed, check out the Monit website (http://mmonit.com/) to know more about the tool and the checks you can perform.

Even if you're using Monit, you should use some kind of external monitoring service like Pingdom (http://pingdom.com/) to be sure that everything is running smoothly at your end.

Securing your web server

If your hosting provider doesn't have an "emergency" console/terminal for the possible issue that the SSH daemon isn't running or isn't accepting connections, you **should not** perform this firewall setup, as you could become blocked from the machine once the firewall is up and starts blocking all ports. Go on at your own risk ☺

Now that the server is up and monitored, we'll add the last touch to it by enabling a firewall and fine tuning the SSH access. During the server install we added the "ufw" package that is a simple interface to the Linux firewall, let's enable it and open the only two ports we need right now, 22 and 80 – you could open 443 if you would use SSL.

As root, run:

```
ufw enable
ufw allow 22
ufw allow 80
```

Now check the firewall status:

```
ufw status
```

The output should be something like this:

```
root@rails-tutorial:/etc/monit/conf.d# ufw status
Status: active
To                         Action      From
--                         ------      ----
22                         ALLOW       Anywhere
80                         ALLOW       Anywhere
```

With the firewall in place, we finally go to the last step on our server setup, disabling SSH access by password. Most of the attacks that happen on Linux boxes are brute force attacks to login as root. People use dictionaries and try as many combinations as possible to login to a machine, once you disable SSH access using password (and allow access only using private/public keys) this attack becomes virtually impossible.

The main caveat on this solution is that if you ever lose your private key or has an emergency to access the server, you won't be able to do it unless your hosting company has an "emergency" console or you're taking the private/public key pair with you wherever you go.

To disable the password login, open the "/etc/ssh/ssh_config" file and uncomment this line:

```
# PasswordAuthentication yes
```

And after uncommenting set the value to "no":

```
PasswordAuthentication no
```

Now restart the SSH server daemon:

```
/etc/init.d/ssh restart
```

Closing thoughts

It was a long way, but now you have an Ubuntu server machine ready for the action and running your rails application. This tutorial is by no means complete or defines the "correct" way of doing things, it's just the way I do my server setups and if you have other best practices, ideas, comments or anything, don't be shy and hit the comments box bellow, I'd love to hear how people are handling and doing their deployments.

If you have questions or ideas about topics on deployments that I should cover, also post them on the comments as this is just the first post about deployments in a series of posts I'm planning to write.

Enjoy!