A Beginner's Guide to

Deploying Ruby & Rails Applications

Bryan Bibat

A Beginner's Guide to Deploying Rails

Bryan Bibat

This book is for sale at http://leanpub.com/deploying-rails-for-beginners

This version was published on 2015-06-16

Contents

Introduction: Beyond Heroku **1**
Who is this book for? 1

Practicing for Linux Deployment **3**
Setting up a practice VirtualBox Virtual Machine 3
Additional Sever Setup 8

Overview of Web Application Server Systems **16**
Serving Static and Dynamic Web Content 16
How to Serve Content from Ruby Applications 18

Installing Phusion Passenger **20**
Installing Ruby 20
Installing nginx + Passenger 21

Deploying Ruby Web Applications to Passenger **25**
A Quick Look at nginx Server Settings 25
Deploying a Simple Sinatra App 26
Deploying a Simple Rails App 31

Redeploying Rails Apps on Passenger **38**
Modifying the Rails App 38
Adding File Upload with Carrierwave 39
Simplifying Deployment with Mina 40

Deploying on a VPS **47**
Overview of Popular VPS Providers 47
Setting Up Your Custom Domain 49

Introduction: Beyond Heroku

"*How do I deploy my Rails app to a server?*" is the inevitable question that comes out of people who have just finished a tutorial on Rails.

Tutorials like *Rails Tutorial* and *The Odin Project* only teach their students to deploy to Heroku. Heroku is a great service, but recent changes have required all free apps to have 6 hours of sleep per day. Any newbie expecting round-the-clock traffic to their app would now consider getting a low-cost virtual private server (VPS) rather than the lowest-cost "never sleeping" Heroku tier.

Those considering the VPS route will have some problems, though. To start off, the official Rails site lacks beginner-friendly documentation, providing only links to software. Also, while there are a lot of tutorials out there that teach you how to setup Rails on VPSs, they usually cover only the "How" of deploying and not the "Why". These tutorials also talk about only one set of technologies; you will have to research on your own if you want to learn other alternatives.

I wrote this book to answer those concerns. At its core, this book is no different from other online tutorials - it will provide you with the steps to deploy your Rails application on an online VPS. In addition, however, I will also try to discuss the reasoning behind all of the steps to avoid the "just do this, trust me" magical thinking trap that many beginners fall into when learning something technical.

This book is split into two parts: the first covers practicing on a local virtual machine up to deploying to a very basic nginx/Passenger server stack. This is the "sample" book that is free to download and read online on Leanpub. The second part covers alternative stacks and more advanced deployment tools. This part is only available for those who have purchased the book.

Who is this book for?

This book targets two main audience groups:

- **Beginners to Ruby and Rails web development** - whether you're considering leaving Heroku for a VPS or you're just curious how Ruby servers are setup, this book is for you.
- **System Administrators new to deploying Ruby web apps** - you've set up dozens of PHP, Java, .NET, etc. servers before and you want to get an idea how things are done in the Ruby ecosystem. This book will help you as long as you don't mind skimming over long stretches of lessons about things that you already know.

This book assumes that:

- You have basic Rails knowledge. For SysAdmins, basic concepts are enough. For Devs, you should at least complete a basic tutorial like *Rails Tutorial*.
- You are comfortable with using the terminal or command line. We will be primarily using *nix commands, but Windows users should also be able to run them under MinGW or Git Bash.

- You have a computer that can run a virtual machine i.e. non-netbook with >2GB RAM and ∼20GB free hard drive space.

This book is ***not*** for the following:

- **Experienced Ruby Devs/SysAdmins** - I mean, you're also free to browse the book, but don't blame me if you feel everything in this book is dumbed down for beginners.
- **Devs with a deadline** - your company wants to roll out their shiny new project next week/tomorrow/**right now** but they won't shell out for Heroku/Engine Yard/a Ruby SysAdmin so they want you, the new guy, to set up the servers for them. Sorry, but this book only covers the basics and shouldn't be the sole resource for serious deployments. It doesn't even go into detail about fully securing your severs. I suggest you negotiate for an extension to give yourself time to learn what you need.
- **People who won't touch Linux** - we will be using the latest "long term support" version of Ubuntu (14.04) as our target production environment. If you're strongly anti-Ubuntu but pro *nix, you'll still learn a lot of things from this book. On the other hand, there's nothing here for die-hard Windows fans that won't touch anything other than Windows. If you absolutely have to deploy on Windows, look for JRuby tutorials on how to convert your app to JRuby and deploy them on Java application servers.

Practicing for Linux Deployment

It's not expensive to practice using a VPS - VPSs are rented for a few cents an hour while some services line Amazon just require a credit card for using their free-tier. However, to avoid potentially expensive accidents like forgetting to destroy a VPS after practice, we will be asking you, the beginner, to practice on Virtual Machines (VM) running on your computer.

Setting up a practice VirtualBox Virtual Machine

Download the following:

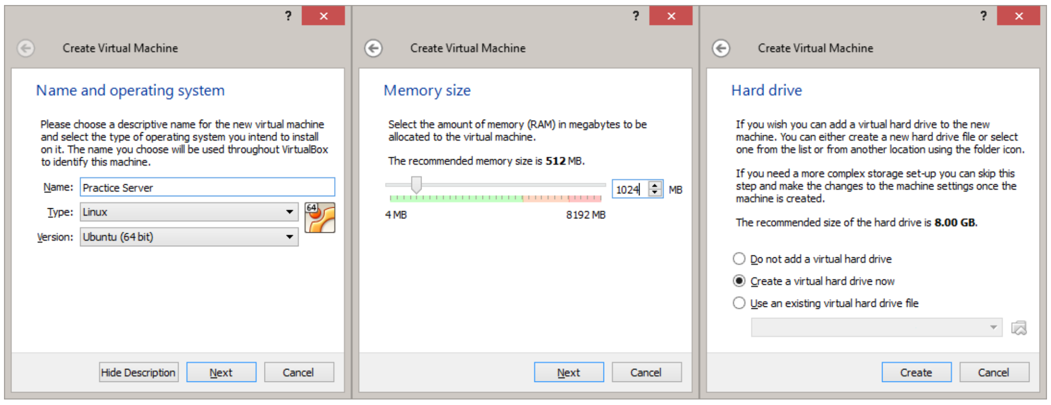
- VirtualBox - get version 4.3.28 or higher for your OS
- Ubuntu Server 14.04 LTS - the download button should give you the latest LTS version (14.04.2) that can be used by your machine (i.e. 64-bit). You can also download a BitTorrent torrent file at the Alternative Downloads page

Install VirtualBox after downloading. It will install a couple of drivers but overall it should be a smooth installation process.

> **VMware**
>
> You can also use VMware Player (Windows) or VMware Fusion (Mac) for this tutorial. Just use the settings equivalent to the ones discussed below.

Creating a new Virtual Machine

Once VirtualBox is installed, you can now create your practice server. Click the "New" button to start the process.


Create Virtual Machine

Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: Practice Server
Type: Linux
Version: Ubuntu (64 bit)

Create Virtual Machine

Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **512** MB.

1024 MB

Create Virtual Machine

Hard drive

If you wish you can add a virtual hard drive to the new machine. You can either create a new hard drive file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard drive is **8.00 GB**.

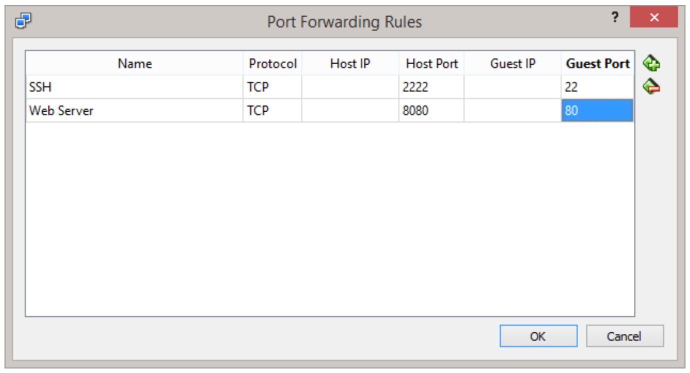
- ○ Do not add a virtual hard drive
- ● Create a virtual hard drive now
- ○ Use an existing virtual hard drive file

- **Name and Operating System** - this step is just for labeling purposes. Choose any name you want and set the Type/Version to Linux/Ubuntu (64 bit).
- **Memory Size** - 512 MB is enough for a practice server. Properly tuned, it may even be good enough for small to medium sized sites. But for this practice, I suggest going for 1 GB to give you some extra leeway.
- **Hard Drive** - The first 3 defaults ("Create a virtual hard drive now", "VDI", "Dynamically allocated") are fine. You may choose to change the location of your virtual hard drive and increase its size in the "File location and size" step. Note that by choosing dynamically allocated storage, the full hard drive space will not be allocated initially, saving you space in your real hard drive.

Your practice server VM has now been created. We still have to modify some settings before we can continue. Highlight the VM and click the Settings button.

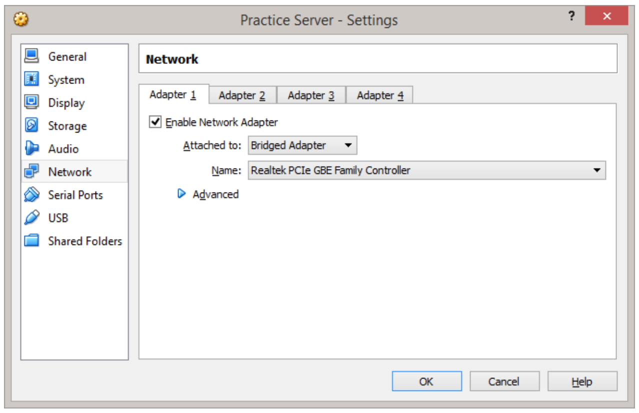
- **Network** - you have two main choices for networking: *NAT* and *Bridged Adapter*

In NAT mode, the VM connects to its own virtual router which uses the internet connection of the host OS. The downside to this is that the VM is invisible to the network and you have to set Port Forwarding settings to allow you to connect to it. Here are the port forwarding settings to use if you prefer NAT:


Port Forwarding Rules

Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
SSH	TCP		2222		22
Web Server	TCP		8080		80

In Bridged Adapter mode, the VM directly uses an attached network adapter on the host OS to connect to the internet. This means that it will also retrieve its own IP address from the router's DHCP server and in turn be visible in the local network so there's no need for port forwarding. Just choose the adapter connected to the network and you're good to go.

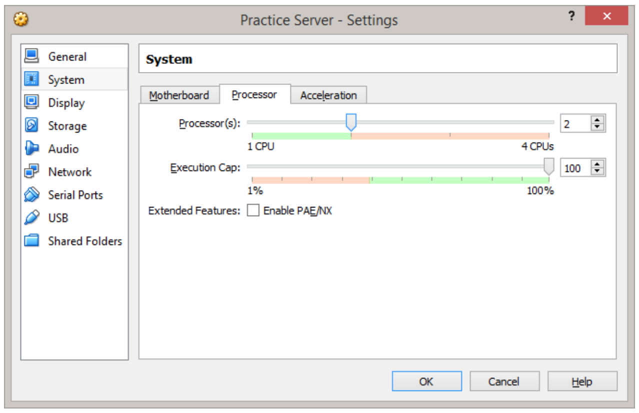


[✓] Enable Network Adapter
Attached to: Bridged Adapter
Name: Realtek PCIe GBE Family Controller

I suggest using Bridged Adapter since it simulates what you'll get from VPSs (i.e. just an IP address for access) but we'll provide instructions for both setups so you can choose either of the two.

- **Storage** - Select the Empty CD and click the CD icon to find the .iso file for Ubuntu Server. The default boot order is Floppy, CD, Hard Drive so there's no need to change that setting.

We should now be ready to install Ubuntu Server but there's one optional setting you can change:

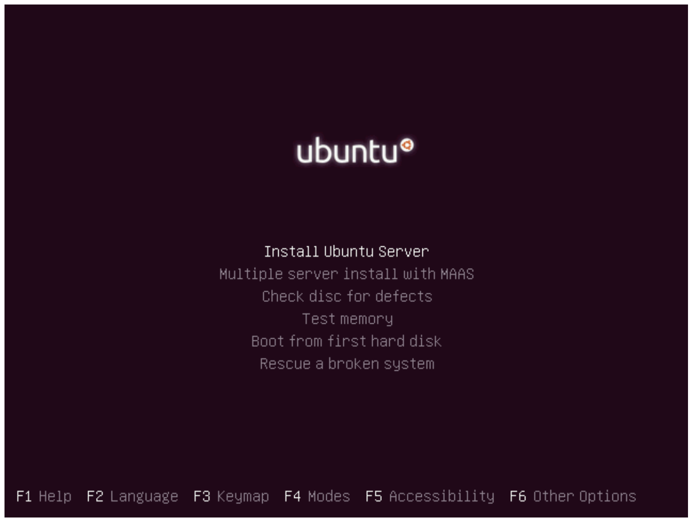


Processor(s): 2
Execution Cap: 100
Extended Features: [ ] Enable PAE/NX

Increasing Processor count not going to improve performance that much, but it may help later when we go into version managers that compile Ruby.

Install Ubuntu Server 14.04

Start the virtual machine and wait for the installer to load. Select your language and proceed to "Install Ubuntu Server".


ubuntu
Install Ubuntu Server
Multiple server install with MAAS
Check disc for defects
Test memory
Boot from first hard disk
Rescue a broken system
F1 Help F2 Language F3 Keymap F4 Modes F5 Accessibility F6 Other Options

Arrow keys and Tab moves the cursor, Enter activates the buttons, and Space selects options.

The mouse pointer may be captured by the VM preventing you from switching to this document. When that happens, press the Host key (Right Ctrl by default) to release the pointer.

- **Select a language** - choose your preferred language
- **Select your location** - this affects the server used for downloading packages so choose your country
- **Configure your keyboard** - unless you're using a localized keyboard, just choose "No", "English (US)", "English (US)"
- **Configure the network** - hostname can be anything. For this tutorial we'll use "server" to make it obvious in the examples
- **Set up users and passwords** - full name, user name, and password can also be anything. For this tutorial, our default user will be "user". Also, choose not to encrypt your home directory.
- **Configure the clock** - choose "No", and scroll all the way down to choose "UTC". This will simulate how some VPSs use UTC as default and will force you to think that servers can have different time zones than your local machine.
- **Partition disks** - this part may be daunting but just all about choosing the defaults and agreeing to prompts: "Guided - use entire disk and set up LVM", "SCSI 3 (0, 0, 0)…", "Yes", "Continue", "Yes"
- **Configure the package manager** - leave this blank unless you're using an HTTP proxy
- **Configuring tasksel** - since this is just a practice server, leave it to "No automatic updates"
- **Software selection** - use Space to select "OpenSSH server" and press Enter to Continue
- **Install the GRUB boot loader on a hard disk** - choose "Yes"
- **Finish the installation** - select "Continue" to eject the installation disk and restart the server

Upon restarting, go ahead and check if you can log in using the user name and password that you provided earlier. If you can't, it may be because you have the Caps Lock on or you may have forgotten your credentials. Do everything all over again if it's the latter.

If you chose Bridged Adapter for your VM's networking settings, enter the following command to determine the IP address of your server:

```
$ hostname -I
```

Command Line Crash Course

If you aren't familiar with the Linux command line, please take this time to run through the Command Line Crash Course inside the VM.

Remotely connect via SSH

From this point on, we will be accessing the VM remotely via Secure Shell (SSH) to simulate how to remotely administer a VPS.

Linux and Mac users can use the terminal, while Windows users can use Git Bash. As mentioned in the Introduction, this book assumes that you are familiar with these terminals, either as someone who has gone through a Rails tutorial, or as a SysAd.

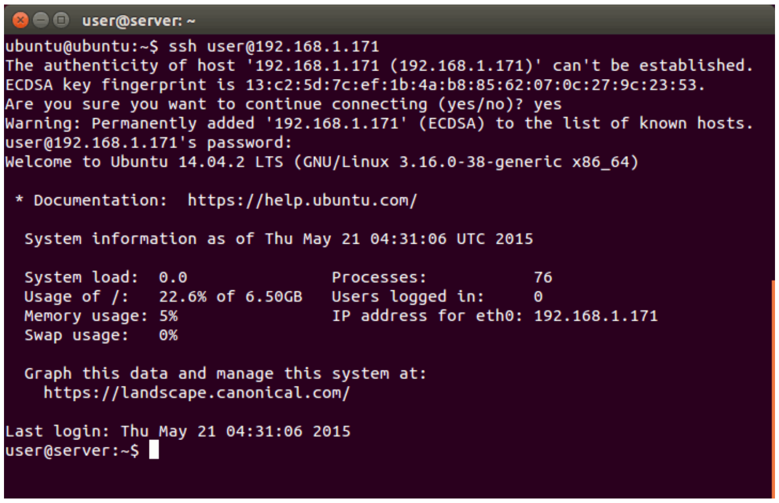
To connect to Bridged Adapter VMs via SSH, run "`ssh [user name]@[server IP address]`" e.g.:

```
$ ssh user@192.168.1.10
```

If you're using NAT with ports forwarded as in the previous section, use "`ssh [user name]@localhost -p 2222`" e.g.

```
$ ssh user@localhost -p 2222
```

Either way, the server will ask you for the user's password. It may also ask you to verify the authenticity of the host (it's local so just say "yes"). Once logged in, you now have full remote control of the server.



```
ubuntu@ubuntu:~$ ssh user@192.168.1.171
The authenticity of host '192.168.1.171 (192.168.1.171)' can't be established.
ECDSA key fingerprint is 13:c2:5d:7c:ef:1b:4a:b8:85:62:07:0c:27:9c:23:53.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.171' (ECDSA) to the list of known hosts.
user@192.168.1.171's password:
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-38-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Thu May 21 04:31:06 UTC 2015

  System load:  0.0               Processes:           76
  Usage of /:   22.6% of 6.50GB   Users logged in:     0
  Memory usage: 5%                IP address for eth0: 192.168.1.171
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

Last login: Thu May 21 04:31:06 2015
user@server:~$
```

Prompt format

To avoid confusing beginners, we will not be using the standard tutorial terminal prompt "`$`". Instead, all of the examples in the book for server commands will use the `bash`'s default prompt format to display the current user and working directory e.g.

```
user@server:~$
```

This prompt is made of the following parts:

- `user` - the username of the current logged in user
- `@` - a separator (i.e. "at")
- `server` - the hostname of the current machine
- `:` - another separator
- ~ - the current working directory. ~ refers to the user's home directory i.e. `/home/user/`
- `$` - this changes to `#` when the current logged in user is `root`

Commands to be executed on your local machine (e.g. Mina commands) will have bare `$` prompts.

Additional Sever Setup

We need to do a few more things before we can proceed with learning how to deploy Ruby and Rails applications.

Updating and Installing New Software

Fresh installs of Ubuntu are usually have packages that are out of date. Let's update them now.

```
user@server:~$ sudo apt-get update
```

As briefly mentioned in the Command Line Crash Course, `sudo` lets you run commands as the `root` user, allowing you to do things that are not permitted for regular users. As an added precaution, using `sudo` requires you to enter the user's password to proceed.

Ubuntu is derived from Debian so it also uses APT (Advanced Packaging Tool) for managing packages. Running `apt-get update` retrieves the latest package lists from the current sources (Ubuntu's official software package repositories by default). It does not upgrade the packages, however. That is done by the next command:

```
user@server:~$ sudo apt-get dist-upgrade
```

This will show you a list of packages that will be updated and will prompt you to continue. Enter "y" or just press Enter (the capitalized letter is the default) to proceed. You can also add the "`-y`" option to automatically yes to all prompts e.g. "`sudo apt-get dist-upgrade -y`".

The command "`apt-get dist-upgrade`" compares the recently retrieved package list with the currently installed packages and upgrades all packages that have new versions. We can also use "`apt-get upgrade`" if we only want to install new packages that don't affect other existing packages (e.g. don't install kernel updates because they uninstall old kernels), but we use the former here since we want to upgrade everything.

Upgrading will take a while to finish. When it's done, let's now install some of the software that we will be using in the apps that we will deploy:

```
user@server:~$ sudo apt-get install build-essential git-core nodejs postgresql libpq-dev -y
```

As the name implies, "`apt-get install`" installs new packages from Ubuntu's repository and other repositories you may have added beforehand. Here's a quick explanation for each of the packages:

- `build-essential` - a package that contains all of the packages needed for compiling and building common applications (e.g. for C, C++)
- `git-core` - core Git packages which we will use later to pull code from repositories
- `nodejs` - NodeJS for Sprockets / Asset Pipeline. It's not up to date, but it should be fine for our needs.
- `postgresql` - our database
- `libpq-dev` - when installing gems, they can sometimes require development libraries to compile against. This package contains the libraries for PostgreSQL and is required for installing the `pg` gem.

Removing Password Login

Let's add the bare minimum level of security to our server.

First off, let's remove password login and replace it with public-key cryptography. That is, we'll tell SSH to reject password logins (which can be brute-forced) and instead verify the user's identity using their private-public key pair.

SSH Keys

Rails tutorials often require you to generate your SSH keys so that you can push your code to GitHub or Heroku. If you haven't generated them yet or are not sure if you have, please go to GitHub's Generating SSH Keys learn how to generate them now.

Start by copying over your public key to the user's authorized keys list. Windows (Git Bash or minGW) users have to do this manually:

```
user@server:~$ mkdir .ssh
user@server:~$ nano .ssh/authorized_keys
```

These commands will create the `.ssh` folder and open the `authorized_keys` file in an editor, in this case `nano`. Open your public key (typically `C:\Users\username\.ssh\id_rsa.pub`) in a text editor (Notepad should be ok), copy it's contents to clipboard, and in the terminal, right-click the window title -> Edit -> Paste. Press Ctrl-O and Enter to save then press Ctrl-X to quit `nano`.

Editors

We will be using `nano` as the default editor in this book. You're free to use other editors like `emacs` or `vim` if you're more comfortable with them. The former isn't installed by default so you will have to install it via "`sudo apt-get install emacs24-nox`".

Windows Command-line QuickEdit Mode

The QuickEdit Mode is a quicker way to paste and even copy text from Windows terminals based on `cmd.exe` like Git Bash. To enable this, right-click the window title -> Properties -> Options -> tick QuickEdit Mode.

Once enabled, you can now right-click the terminal to paste your clipboard. You can also highlight and right-click to copy text from the terminal.

We also need to reduce the permissions of the directory and the file. Otherwise, the SSH server will consider them insecure and ignore them.

```
user@server:~$ chmod 700 .ssh
user@server:~$ chmod 600 .ssh/authorized_keys
```

The `chmod` command changes file and directory permissions. We're using octal mode here so it's not obvious what `700` or `600` means, but basically `chmod 600` means that only the user can read and write the file while `chmod 700` means that only the user can read, write, and execute it.

Linux and Mac users have a shortcut that does the whole process. Open a new terminal or `exit` your current session and run the following:

```
$ ssh-copy-id user@[IP address]
```

If you're using NAT and port forwarding:

```
$ ssh-copy-id user@localhost -p 2222
```

Mac users may have to first install ssh-copy-id via Homebrew i.e. “`brew install ssh-copy-id`”.

Now that you’ve added your public key to the server, open the SSH’s configuration file and change its settings.

```
user@server:~$ sudo nano /etc/ssh/sshd_config
```

There are two things we need to change. One is to disable root login:

```
# Authentication:
LoginGraceTime 120
~~PermitRootLogin without-password~~
**PermitRootLogin no**
```

And the other is to remove password authentication and force logging in via private-public key pair:

```
# Change to no to disable tunnelled clear text passwords
~~#PasswordAuthentication yes~~
**PasswordAuthentication no**
```

Once done, you can restart the SSH server daemon to apply the changes for all new connections:

```
user@server:~$ sudo service ssh restart
```

Adding a Regular User for Deployment

As another security precaution, we will be running our Ruby and Rails applications under a user without any administration rights. Enter the following command to create this user:

```
user@server:~$ sudo adduser --disabled-password --gecos 'Deploy User' deploy
```

This command creates the “`deploy`” user and its corresponding “`deploy`” user group. The options we added remove disables the `deploy` user’s password (but still allows login via private-public key) and sets the user information to “Deploy User” (otherwise we’ll be prompted to enter it). This command also creates the user’s home directory “`/home/deploy/`”.

To allow us to login via SSH, let’s copy over our ssh settings from our `user` user to the `deploy` user.

```
user@server:~$ sudo cp -r .ssh /home/deploy/.ssh
user@server:~$ sudo chown -R deploy:deploy /home/deploy/.ssh
```

The `chown` command changes the ownership of the folder; in this case, we changed the ownership of the folder and all of its contents (`-R`) to both the `deploy` user and user group.

If setup correctly, you should be able to login remotely to the `deploy` user e.g.:

```
$ ssh deploy@192.168.1.10
```

Or:

```
$ ssh deploy@localhost -p 2222
```

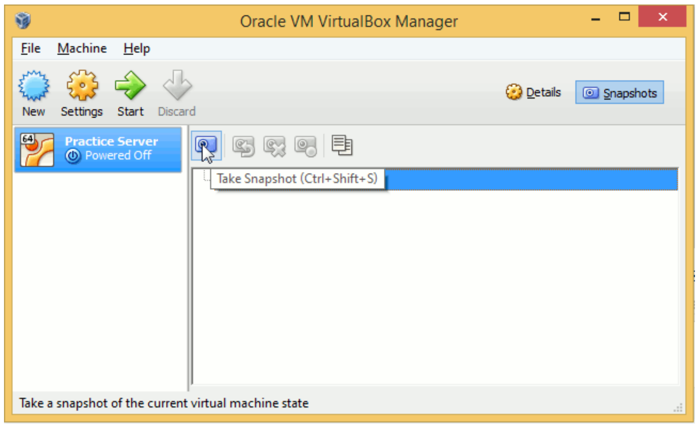
Wrap-up: Shutdown server and take a Snapshot

You should now be ready to start installing the software required for Ruby and Rails application servers. But before that, let's take a snapshot

Start by shutting down the server:

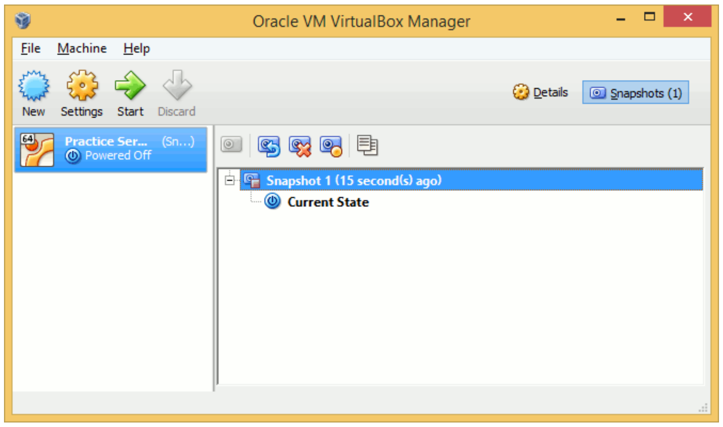
```
user@server:~$ sudo poweroff
```

When the server has shutdown, go back to the VirtualBox Manager, select your practice VM, go to the Snapshots tab, then click the Take Snapshot button.



Take a snapshot of the current virtual machine state

Choose a name for the snapshot and continue.



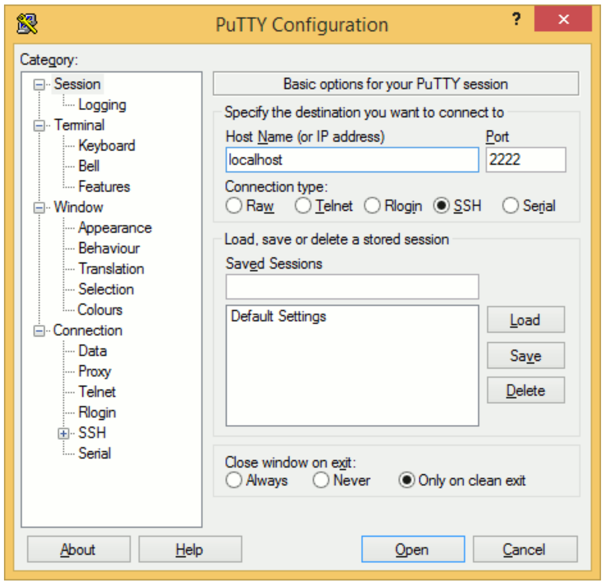
You now have a snapshot of the VM. You should now be able to roll back your server in case you mess up the installation process in the following chapters.

Alternative SSH Client for Windows: PuTTY

While you can certainly go through this tutorial using SSH under Git Bash, the *de facto* standard remote shell client for Windows is **PuTTY**. The latter has more features, is more customizable, and defaults to QuickEdit-style behavior.

You can get PuTTY from this link. We only need PuTTY and PuTTYgen, but download and install the Windows installer is more convenient.

Opening a session is pretty straightforward, just enter the address and the port and press Open to connect. Here's the settings for connecting to a VM with a NAT network:



Specify the destination you want to connect to

Host Name (or IP address): localhost

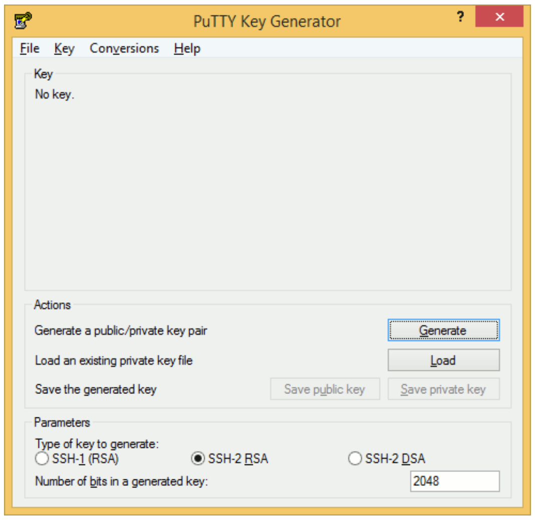
Port: 2222

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Close window on exit: Always Never [✓ Only on clean exit]

You can also save your session settings by entering a name under the Saved Sessions text field and pressing Save.

You will not be able to login yet since we already disabled password authentication and PuTTY does not use the same private-public key format as the one used in the Git Bash's SSH client. We need to convert our private key first using the PuTTYgen program.



No key.

Generate a public/private key pair

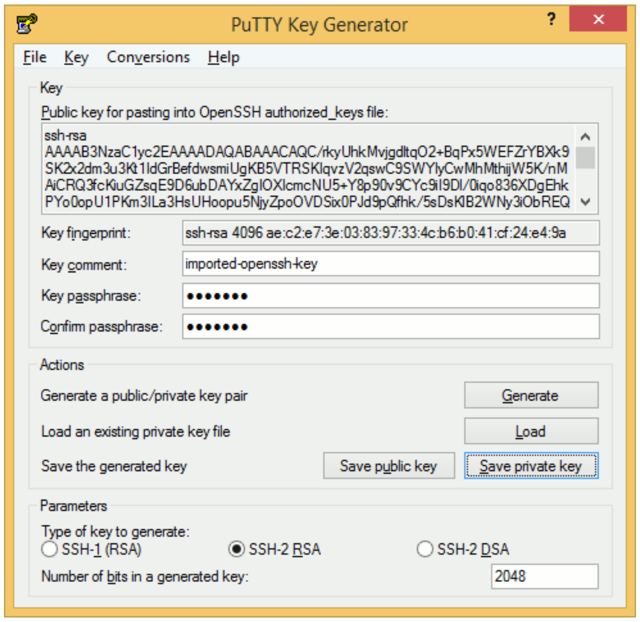
Load an existing private key file

Save the generated key

Type of key to generate:
SSH-1 (RSA) [✓ SSH-2 RSA] SSH-2 DSA

Number of bits in a generated key: 2048

Upon opening PuTTYgen, you will be greeted by a blank form. Here we can generate new private-public key pairs, but since we already have one, we're just going to convert it to PuTTY's format (`.ppk`). Click the Load button (or go to File -> Load Private Key) then change the dropdown from "PuTTY Private Key Files (*.ppk)" to "All Files (*.*)" and open your private key (typically `C:\Users\username\.ssh\id_rsa`).



Public key for pasting into OpenSSH authorized_keys file:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQC/rkyUhkMvjgdltqO2+BqPx5WEFZrYBXk9
SK2x2dm3u3Kt1IdGrBefdwsmiUgKB5VTRSKlqvzV2qswC9SWYlyCwMhMthijW5K/nM
AiCRQ3fcKiuGZsqE9D6ubDAYxZglOXlcmcNU5+Y8p90v9CYc9il9DI/0iqo836XDgEhk
PYo0opU1PKm3lLa3HsUHoopu5NjyZpoOVDSix0PJd9pQfhk/5sDsKlB2WNy3iObREQ
```

Key fingerprint: ssh-rsa 4096 ae:c2:e7:3e:03:83:97:33:4c:b6:b0:41:cf:24:e4:9a

Key comment: imported-openssh-key

Key passphrase: •••••••

Confirm passphrase: •••••••

Generate a public/private key pair

Load an existing private key file

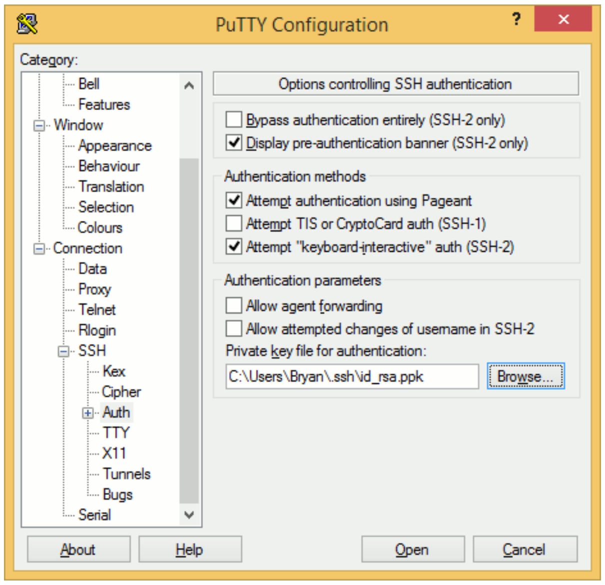
Save the generated key

Type of key to generate:
SSH-1 (RSA) [✓ SSH-2 RSA] SSH-2 DSA

Number of bits in a generated key: 2048

You may be prompted to enter your private key's passphrase in order to continue.

You can now save the private key to the `.ppk` format with the Save private key button. Once you have the private key in the proper format, add it to your PuTTY SSH session settings by going to Connection -> SSH -> Auth -> Browse...



Options controlling SSH authentication

- [ ] Bypass authentication entirely (SSH-2 only)
- [✓] Display pre-authentication banner (SSH-2 only)

Authentication methods

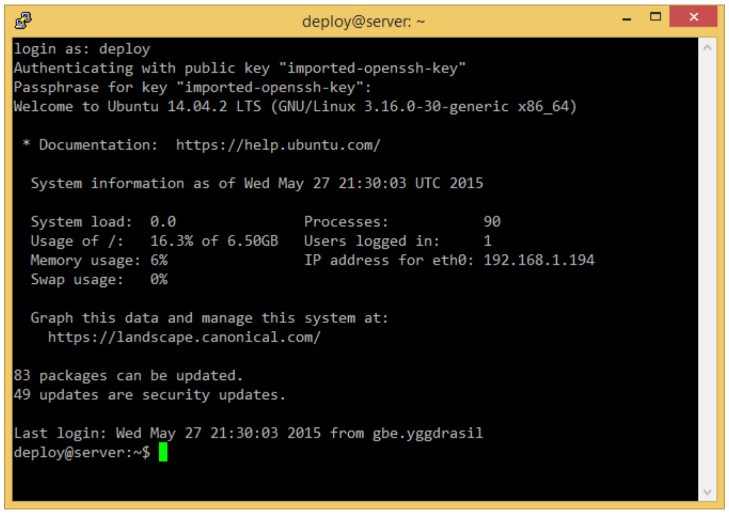
- [✓] Attempt authentication using Pageant
- [ ] Attempt TIS or CryptoCard auth (SSH-1)
- [✓] Attempt "keyboard-interactive" auth (SSH-2)

Authentication parameters

- [ ] Allow agent forwarding
- [ ] Allow attempted changes of username in SSH-2

Private key file for authentication: C:\Users\Bryan\.ssh\id_rsa.ppk

Go back to Session and save so that you won't need to do it again. You should now be able to connect to both `user` and `deploy` users using these settings. Note that as with the normal OpenSSH SSH client, you will be asked first to confirm the identity of the server.

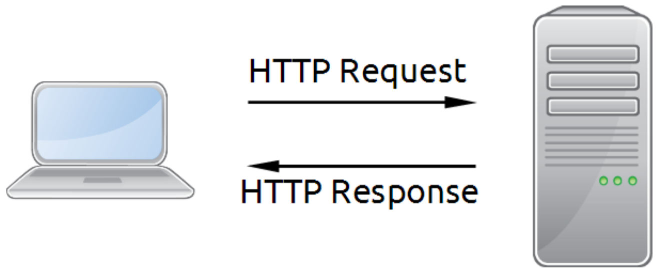


```
login as: deploy
Authenticating with public key "imported-openssh-key"
Passphrase for key "imported-openssh-key":
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-30-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Wed May 27 21:30:03 UTC 2015

  System load:  0.0               Processes:           90
  Usage of /:   16.3% of 6.50GB   Users logged in:     1
  Memory usage: 6%                IP address for eth0: 192.168.1.194
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

83 packages can be updated.
49 updates are security updates.

Last login: Wed May 27 21:30:03 2015 from gbe.yggdrasil
deploy@server:~$
```

connected via PuTTY, font changed to Consolas

Overview of Web Application Server Systems

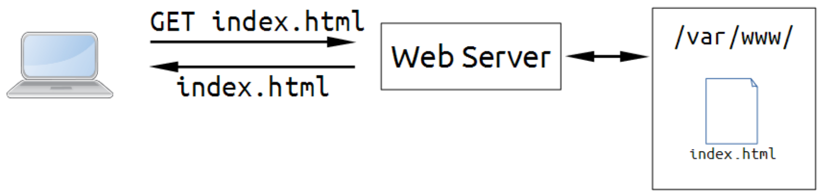
As a developer who has gone through a Rails tutorial or as a SysAd, you should be aware of the HTTP request-response cycle i.e. how a client (e.g. browser) communicates with a server.



But there is more to serving web content than this simple diagram. So before we begin, it's best that you at least have an idea what goes on behind web servers.

Serving Static and Dynamic Web Content

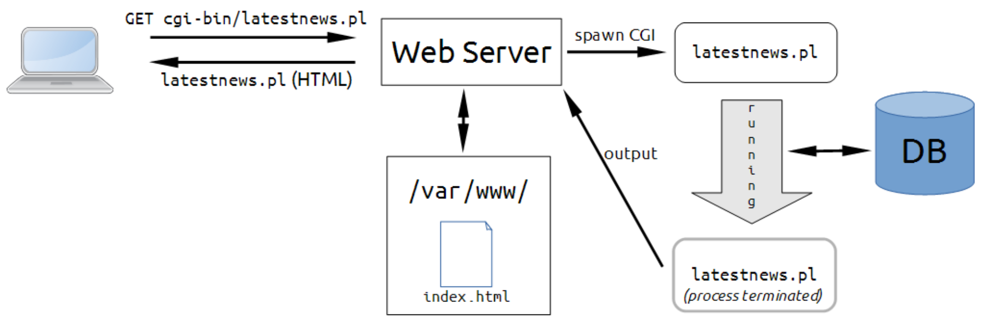
From the early days of the Internet and even up to today, a good portion of the data sent over the internet is static content e.g. images, static text files like CSS, HTML. And that's what a web server does.



In this diagram, the web server program is set to serve files from the `/var/www/` folder. If the requested file exists, it returns it. Otherwise, it returns a "`404 Not Found`" HTTP error.

Eventually the internet became more and more complex that there came a need to serve dynamic content. Say you have a constantly updated news site - manually crafting an HTML page for every news item is difficult enough, what if you need to change the layout of all of the pages? How do you implement article searching?

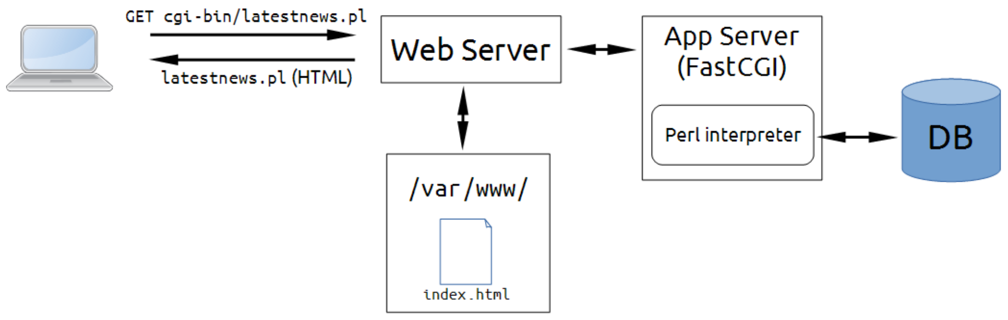
These problems can be solved by saving your data in a database, then dynamically crafting a new page for every request. The simplest way to do this is through CGI (Common Gateway Interface).



In this diagram, the client wants to request the latest news articles and does so by requesting `/cgi-bin/latestnews.pl`. The server has set `/cgi-bin/` folder as the CGI directory so instead of serving the contents of that file, it runs that program (in this case a Perl script) and returns the output to the client. The script can do whatever it wants: it can access the database, read files, send network requests, etc., all the server cares about is its output.

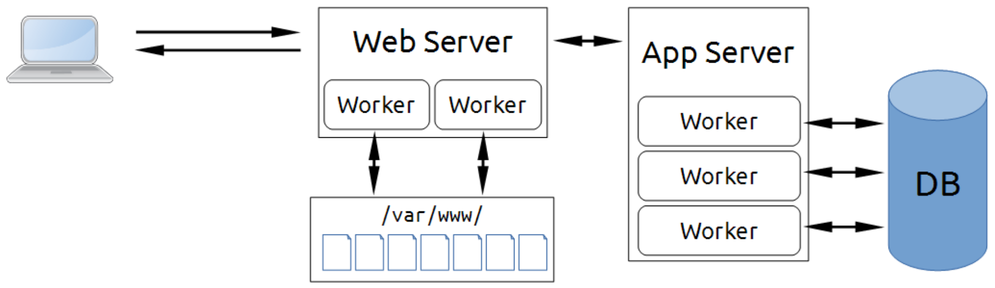
The simplicity of CGI made it popular but it had one main problem: starting up programs and killing them every request incur a significant overhead.

One possible solution is to have a process that loads the interpreter and keeps it running while it accepts requests for dynamic content from the web server. This is basically what FastCGI does.



Here we see the web server which still serves static content. It is also configured to pass the requests for dynamic content to the separate FastCGI server - our new *application sever*. Properly configured, the request to `/cgi-bin/latestnews.pl` will return the same result as the CGI setup, only faster.

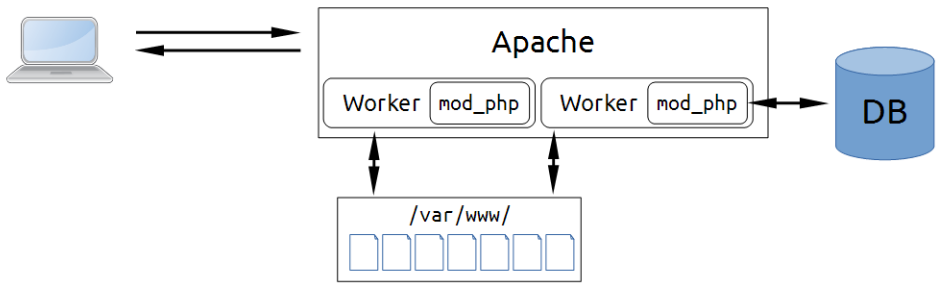
The diagram above is oversimplified. Here's a slightly more detailed take on the system:



Your web and application servers aren't limited to being single-process systems. Each can have multiple processes and threads in order to handle multiple loads concurrently. In turn, those workers are managed by the master process: the master can spawn new workers to handle increased load, balance the requests between workers, kill non-responsive workers, and so on.

Another thing about the FastCGI-style: the web and application server are separate processes and have to use either TCP sockets or UNIX sockets to communicate with each other.

Of course, the FastCGI-style isn't the only alternative to CGI. For example, a popular solution for PHP is to use Apache HTTP Server (commonly shortened to just "Apache") and enable the `mod_php` module:

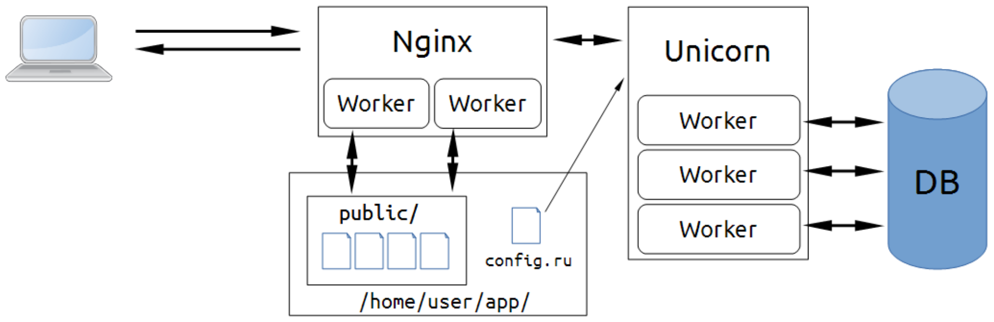


This loads the PHP interpreter to all of the workers allowing Apache to process PHP scripts without having to use a separate application server, essentially making it a combined web and application server.

How to Serve Content from Ruby Applications

All modern Ruby web applications use Rack as an interface between the application and the application server. Any application server that supports Rack will be able to load a Ruby web application with Rack and use the latter to serve requests.

If you look at your Ruby and Rails apps, you will see the standard entry points for both the web and the Rack-enabled application server: the `/public` folder often refers to the root directory that will be served by the web server, while the `config.ru` (**r**ack**u**p configuration) is what the Rack server will look at for starting the Ruby application. Here's an example with Nginx and Unicorn serving a Rails application:



Nginx at the front, serving precompiled assets and other static files from the `/home/user/app/public/` folder. Other requests will be directed to Unicorn, which by default runs the Rails application through `/home/user/app/config.ru`.

As the diagram implies, Unicorn is a multi-process application server separate from the web server. It is similar to the FastCGI server above, but we can't call them FastCGI servers since it doesn't follow the FastCGI protocol. (With some effort, you *can* use FastCGI servers to serve Rack applications, but it's definitely not recommended.)

Other Rack application servers follow the same theme with some differences. Passenger and Puma can be used as multi-process servers, and they can also be used as multi-threaded servers when used with an interpreter that supports real threading like JRuby. Thin and Rainbows are servers which use a different approach (evented) to improve concurrency in certain types of applications.

In this tutorial, we will be starting off with Passenger and Nginx. When installed as an Nginx or Apache module, Passenger makes them behave like combined web and application servers - instead of having to configure two different servers separately, all configuration is done in the Nginx/Apache side.

Installing Phusion Passenger

In this chapter, we will install nginx (pronounced "engine x") with Phusion Passenger, our combined web and application server. We will also install Ruby 2.2 which will be used by Passenger when spinning up the application workers.

Most of the commands here require `sudo`; we'll work around this minor annoyance by switching to the `root` user.

```
user@server:~$ sudo -i
```

This will switch you from the `user` user to the `root` user and move you to `root`'s home directory, "`/home/`".

Installing Ruby

First we'll install Ruby. We'd like to install it via `apt-get` again, but unfortunately the version in Ubuntu's official repository for Ubuntu 14.04 is out of date (version 1.9.3p484 as of this writing).

Good thing we can add third party repositories to APT, in this case Brightbox's PPA in Launchpad.

```
root@server:~# apt-add-repository ppa:brightbox/ruby-ng -y
```

After adding the repository, we must retrieve the new package list by running `apt-get update` again:

```
root@server:~# apt-get update
```

Now to finally install install Ruby 2.2. We also include the development libraries since they are required when installing gems.

```
root@server:~# apt-get install ruby2.2 ruby2.2-dev -y
```

The Ruby installed via the Brightbox repository is separate from the one available in Ubuntu's official repository but it will not be in conflict even when installed side by side: upon installation, the default behavior of both `ruby` and `gem` commands will be changed to use the newly installed version. If you want to use the old version, you can call the appropriate executable directly (e.g. `/usr/bin/ruby1.9.1`) or you can install and use the `ruby-switch` package also available in the Brightbox repository.

Our last step in installing Ruby would be to install Bundler:

```
root@server:~# gem install bundler --no-ri --no-rdoc
```

Why not use RVM, rbenv, etc?

Many tutorials out there will tell you to install a version manager of some sort for your production server. However, when you think about it, you'll realize that ***you don't need them***.

Let's look at the main reasons why we use version managers:

- *System Rubies (e.g. from Ubuntu's repos) are usually out of date and we need version managers to install them* - not a problem since we can install the latest version of Ruby via 3rd party repositories like Brightbox's
- *You can install gems without having root access* - after installing Bundler, you can install gems on a non-root path via the `--path` option and you run them using `bundle exec`. And since binstubs are enabled by default in Rails 4, you don't need to type out `bundle exec` much anymore.
- *Version managers allow us to switch between Ruby versions* - you usually only have one app installed per server in a production setting.

Version managers have their drawbacks; we just don't realize them because they're necessary in development environments:

- *Compiling Ruby takes time* - we have no choice in development but to let our version manager compile and install a new Ruby to our home directory, and this can take around 10 minutes for a decent computer. This may be a non-issue if you have a top of the line server, but I still doubt it can beat the less than one minute installation speed of installing via APT.
- *Path craziness* - version managers use different techniques to modify the `PATH` to avoid conflicting with the system Ruby and to auto-switch between versions. As mentioned above, we don't need to switch Ruby versions and these techniques can complicate certain aspects of your system e.g. job scheduling via `cron`.

Of course, we cannot ignore that there are still use cases for version managers so we'll cover how to use them in a latter chapter.

Installing nginx + Passenger

Next up is installing our web and application servers, nginx and Passenger. Like Ruby 2.2, we will install both using `apt`, but this time the setup will be slightly different.

The Passenger package is not hosted in Launchpad, so instead of `apt-add-repository` we'll have to manually get the authentication key and add the URL of the Passenger repository.

Run the following command to get the key which will be used by `apt` to authenticate the Passenger packages:

```
root@server:~# apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 561F9B9CAC40B2F7
```

Let's then add the APT repository for Ubuntu 14.04 to `apt`'s sources:

```
root@server:~# echo "deb https://oss-binaries.phusionpassenger.com/apt/passenger trusty main" \
  > /etc/apt/sources.list.d/passenger.list
```

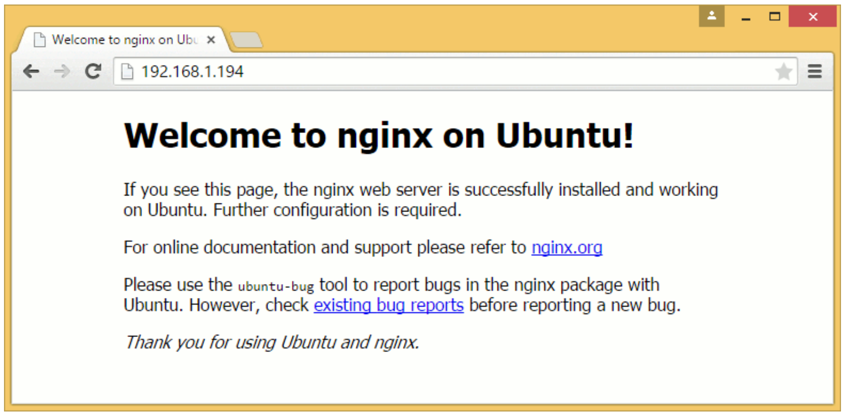
(We split the command to multiple lines with a backslash (\) because the command is too long for this document.)

The `echo` command outputs a string to the console, which is then redirected by `>` to the new file `/etc/apt/sources.list.d/passenger.list`. You can do the same by creating and editing the file via a text editor (e.g. `nano /etc/apt/sources.list.d/passenger.list`) but the command above has fewer steps.

And finally we install nginx and Passenger:

```
root@server:~# apt-get update

...

root@server:~# apt-get install nginx-extras passenger -y
```

By now you should be able to verify that nginx is installed by opening your browser to the URL of your virtual machine (or `http://localhost:8080/` if you're using NAT and port forwarding).



Welcome to nginx on Ubuntu!

If you see this page, the nginx web server is successfully installed and working on Ubuntu. Further configuration is required.

For online documentation and support please refer to nginx.org

Please use the `ubuntu-bug` tool to report bugs in the nginx package with Ubuntu. However, check existing bug reports before reporting a new bug.

Thank you for using Ubuntu and nginx.

Why not install via passenger gem (i.e. passenger-install-nginx-module)?

Same reason as why we're installing Ruby via packages: it's just more convenient especially for beginners. Installing via the `passenger` gem requires more commands to be entered, has a 2-5 minute compilation step, and needs more configuration to make it start automatically on startup. Not to mention that you have to go through all that again when you upgrade Passenger.

Install via packages whenever possible. Your SysAds will thank you for that later.

Wrap-up: Enable Passenger

Open nginx's configuration file `/etc/nginx/nginx.conf` with your favorite editor e.g.

```
root@server:~# nano /etc/nginx/nginx.conf
```

You should see the default settings for nginx:

```
user www-data;
worker_processes 4;
pid /run/nginx.pid;

events {
        worker_connections 768;
        # multi_accept on;
}

http {

        ##
        # Basic Settings
        ##

        sendfile on;
        tcp_nopush on;
...
```

Note that most of the file is enclosed in a `http { }` block. This is how settings are applied in nginx - each setting is a line ending in a semicolon, settings are applied to the current enclosing block, and certain blocks can be nested within each other.

To enable Passenger within nginx, we must uncomment the following lines inside the `http` block.

```
        ##
        # Phusion Passenger config
        ##
        # Uncomment it if you installed passenger or passenger-enterprise
        ##

~~        # passenger_root /usr/lib/ruby/vendor_ruby/phusion_passenger/locations.ini;~~
~~        # passenger_ruby /usr/bin/passenger_free_ruby;~~
        passenger_root /usr/lib/ruby/vendor_ruby/phusion_passenger/locations.ini;
        passenger_ruby /usr/bin/passenger_free_ruby;
```

We don't need to modify Ruby's path because `/usr/bin/passenger_free_ruby` is practically an alias for the default Ruby i.e. the one we installed from Brightbox earlier.

Now let's restart the server to apply the changes. As pointed out above, had we installed nginx and Passenger manually, we would have had to also manually install System V init scripts that will start the server for us on server startup. But since we installed via packages, the init script is already installed for us at `/etc/init.d/nginx`. We can call `service` to run this script and restart the server for us:

```
root@server:~# service nginx restart
```

Open the server again through your browser. You should see the same page again if you properly set the `passenger_root` and `passenger_ruby` settings and not an error page.

We will discuss how to configure the other blocks in the next chapter.

Deploying Ruby Web Applications to Passenger

In this chapter, we will be discussing how to configure nginx and Passenger to serve Ruby web applications. Before that, let's look at how nginx is currently configured to serve static files.

A Quick Look at nginx Server Settings

The settings for the page we viewed in the browser are not present in the `/etc/nginx/nginx.conf` file. They are instead found in the `/etc/nginx/sites-enabled/` directory and are included by the following line:

```
        include /etc/nginx/sites-enabled/*;
```

Let's look at the only file in that directory, `default`:

```
root@server:~# nano /etc/nginx/sites-enabled/default
```

```
##
# You should look at the following URL's in order to grasp a solid understanding
# of nginx configuration files in order to fully unleash the power of nginx.
# http://wiki.nginx.org/Pitfalls
# http://wiki.nginx.org/QuickStart
# http://wiki.nginx.org/Configuration
#
# Generally, you will want to move this file somewhere, and start with a clean
# file but keep this around for reference. Or just disable in sites-enabled.
#
# Please see /usr/share/doc/nginx-doc/examples/ for more detailed examples.
##

# Default server configuration
#
server {
        listen 80 default_server;
        listen [::]:80 default_server;

        # SSL configuration
...

        root /usr/share/nginx/html;
        # Add index.php to the list if you are using PHP
        index index.html index.htm index.nginx-debian.html;

        server_name _;
...
```

Read the links in the files comments when you have the time. For now, here's a quick rundown of the settings (*aka* directives) in the above server block:

- **listen** - sets the address and port that the current server is listening to: `listen 80 default_server` means listen to port 80 (default HTTP port) and make this server block be the default server in case a request comes in that doesn't match a `server_name` in other server blocks. The second `listen` line is the same, but for IPv6 addresses.
- **root** - the document root of the server
- **index** - if the request asks for a folder, the files listed will be the ones displayed to the user. For example, when we requested for the "/" path, nginx returned to us the contents of `/usr/share/nginx/html/index.html`.
- **server_name** - this line determines if the request is for this block by matching it with the `Host` header. The dummy entry _ is used here to allow the `default_server` setting in the `listen` directive to catch all non-handled requests.

One thing to note about this server configuration file is merely a *symbolic link*, a special file referencing another file:

```
root@server:~# ls -l /etc/nginx/sites-enabled/
total 0
lrwxrwxrwx 1 root root 34 May 31 20:41 default -> /etc/nginx/sites-available/default
```

As you can see, the real file is located at `/etc/nginx/sites-available/`.

This whole scheme (separating the server settings out of `nginx.conf`, storing those settings in a folder and symlinking it) is not necessary but it makes modifying the files easier later on, especially if you're hosting many servers in a single nginx installation.

(The symlinking scheme is also for Apache users who are more familiar with setting up Virtual Hosts in that manner.)

Deploying a Simple Sinatra App

To better understand how Passenger is configured to start and serve Rack applications, we are not going to go directly to a Rails app. Instead, we will be practicing first on a Sinatra application.

Half of the things we will do in this section will be done as the `deploy` user. We recommend opening a new SSH session under the `deploy` user separate from the `root`.

If you prefer to use just one terminal, you can always switch from `root` to `deploy` via the `su` command:

```
root@server:~# su -l deploy
deploy@server:~$
```

The `-l` option makes the switch behave like it was a login, similar to how `sudo -i` does this for `root`.

To go back to `root`, simply `exit` the session.

```
deploy@server:~$ exit
logout
root@server:~#
```

Setting up and Creating the Sinatra Application

Let's start by creating the project folders.

```
deploy@server:~$ mkdir -p sinatra_demo/public
```

Here `sinatra_demo` is our project folder. The minimal required for a Rack app in Passenger are the `config.ru` file and the `public` folder, the latter of which we created here.

Passenger can use Bundler if it sees a Gemfile. This is our preferred behavior since we do not want to install other gems on the system-level.

Change directories and create a new Gemfile

```
deploy@server:~$ cd sinatra_demo
deploy@server:~/sinatra_demo$ nano Gemfile
```

```
source "https://rubygems.org"

gem "sinatra"
```

And now let's install the Sinatra gem and all required gems to the `bundle` folder using the `--path` option:

```
deploy@server:~/sinatra_demo$ bundle install --path=bundle
```

Now let's create our Sinatra application. Sinatra apps are small so we'll just put all of it inside our `config.ru` file:

```
deploy@server:~/sinatra_demo$ nano config.ru
```

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

set :environment, ENV['RACK_ENV'].to_sym
disable :run, :reload

get "/" do
  "Hello World!"
end

run Sinatra::Application
```

Modifying nginx Settings

Now that we've created our Sinatra application, let's switch back to the `root` user to setup its settings.

First, let's delete the default symlink:

```
root@server:~# rm /etc/nginx/sites-enabled/default
```

Then create the server config file:

```
root@server:~# nano /etc/nginx/sites-available/sinatra
```

```
server {
        listen 80 default_server;
        listen [::]:80 default_server;
        server_name _;

        root /home/deploy/sinatra_demo/public;
        passenger_enabled on;
}
```

If you recall our previous lesson on web and application servers, we needed to point nginx and Passenger to the document root and the application respectively. For nginx, the `root` line does this. On the other hand, we didn't need to tell Passenger where `config.ru` is - it can already guess that it's in the parent directory of the `public` folder.

Overall, the settings here are not much different from the previous nginx static settings apart from the new `passenger_enabled` directive, which, as the name implies, enables Passenger.

All that's left to do is create a symlink:

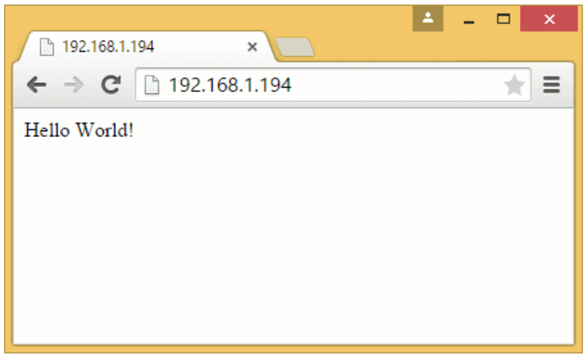
```
root@server:~# cd /etc/nginx/
root@server:/etc/nginx# ln -s ../sites-available/sinatra sites-enabled/sinatra
```

(We changed directories and tweaked the `ln` command so that the line will fit in this document. The full correct command is "`ln -s /etc/nginx/sites-available/sinatra /etc/nginx/sites-enabled/sinatra`".)

And restart the server:

```
root@server:/etc/nginx# service nginx restart
```

Opening the browser should now give you the "Hello World" served from Sinatra.



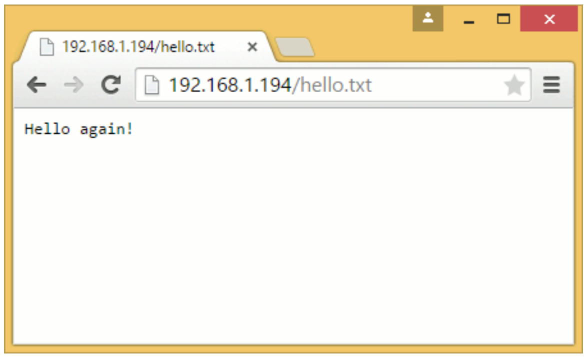
Hello World!

Modifying and Restarting the Application

Let's try modifying our application. What will happen if we added a new file to the `public` folder?

```
deploy@server:~/sinatra_demo$ echo "Hello again!" > public/hello.txt
```

Open `hello.txt` on your browser.

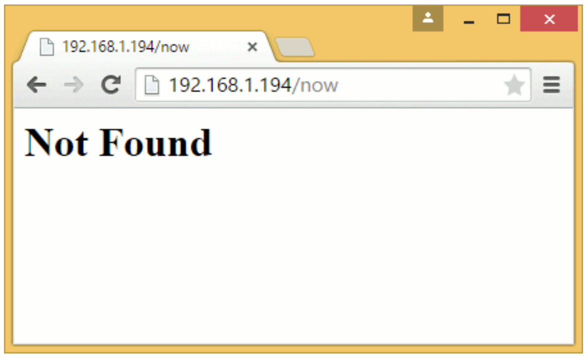


```
Hello again!
```

nginx was able to find the new file and served without problem. But what if we modified our Sinatra application to add a new path handler?

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

set :environment, ENV['RACK_ENV'].to_sym
disable :run, :reload

get "/" do
  "Hello World!"
end

get "/now" do
  Time.now.to_s
end

run Sinatra::Application
```

One might expect that the change will be applied immediately. Unfortunately we get this error:



Not Found

This happens because Passenger caches your entire application on startup to improve performance. This means that all changes to our Rack applications must require an application restart.

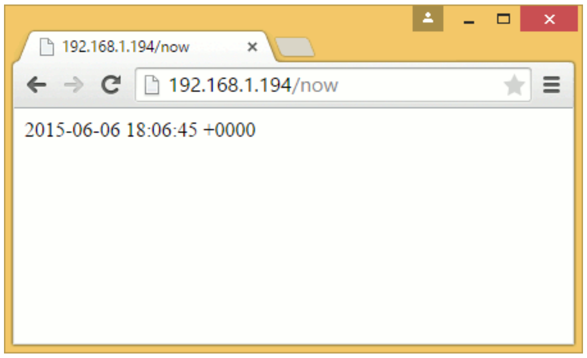
Our `deploy` user doesn't have rights to restart nginx, but there is a way to restart Passenger on the application level: by creating or modifying the `tmp/restart.txt` file.

Let's create the file now to restart our application:

```
deploy@server:~/sinatra_demo$ mkdir tmp
deploy@server:~/sinatra_demo$ touch tmp/restart.txt
```

The `touch` command either updates the timestamp of the file (i.e. touching the file) or creates a new file if it doesn't exist.

Try opening the `/now` page again:



2015-06-06 18:06:45 +0000

Deploying a Simple Rails App

We finally get to the part most of you are here for: deploying a Rails application.

We will be using a sample application for this tutorial. This is be a very simple app, but the concepts that you will learn from it should prepare you for deploying your own app.

Setting up the PostgreSQL Database

Our first step in deploying would be to set up our database user.

In other tutorials, this part also includes setting the user's password. But since we have our PostgreSQL database on the same machine as the application server, there is a simpler alternative, one that we can see in the database's authentication settings:

```
root@server:~# cat /etc/postgresql/9.3/main/pg_hba.conf
# PostgreSQL Client Authentication Configuration File
# ===================================================
#

...

# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

The last two non-comment lines enable md5 password authentication, something that we expect in a database server. On the other hand, the first 2 lines enable "Peer" authentication (aka "Ident" authentication in older versions).

In peer authentication, the database checks the current user name if it is a database user name and allows access as that database user if it does. This authentication does not require a password for login.

One might think that this passwordless scheme is insecure, but it's not that much different from password authentication - in the latter you'll still have the database credentials visible to the current user. And since peer authentication is local-only, remote attacks on your database will require the attacker to have access to the user (if they *do* get access to your user, you've got worse problems than them getting into your database).

We first need to login as the `postgres` (which is essentially the top-level admin user) user in order for us to create our `deploy` database user.

```
root@server:~# su -l postgres
postgres@server:~$ createuser --superuser deploy
postgres@server:~$ createdb deploy
postgres@server:~$ exit
logout
root@server:~#
```

As the command implies, `createuser --superuser` creates our `deploy` user which is a superuser so it has rights to create databases.

Creating the `deploy` database is optional; we added it because many applications assume that a database with the same name as the user exists when using peer authentication.

We log out in the end to go back to our `root` user.

Database Super User

Normally, it would be dangerous to have our database user as a super user. But given our simple setup, the increased privileges on a password-less account doesn't pose that much of a security risk.

If you want, you can use a non-super user account by removing the `--superuser` option in the command. This will prevent you from creating databases later with `rake db:create` and you'll have to create them manually e.g.

```
root@server:~$ sudo -u postgres createdb sample_app
```

Initial Setup for the Rails App

Check out the whole app. It's a public repository so we don't need credentials yet. Later on we'll discuss how to check out code from private repositories.

```
deploy@server:~/sinatra_demo$ cd ~
deploy@server:~$ git clone https://github.com/bryanbibat/sample_app.git
```

Next up is installing the required gems.

```
deploy@server:~$ cd sample_app
deploy@server:~/sample_app$ bundle install --without development test --deployment
```

The first option is self-explanatory: it doesn't install gems in the development and test groups of the Gemfile.

The other option, `--deployment`, is slightly more complicated; it installs the gems for deployment i.e. production servers. More specifically:

- It requires a `Gemfile.lock` to determine the exact versions of the gems to be installed
- Install all the gems to the `vendor/bundle` folder. This is similar to how we installed the Sinatra-related gems to the `bundle` folder.

Production Server Settings

Some application settings are intentionally left out of the code. Database credentials are one; other people should not be able to log in to your production database just because you put your app in a public git repository.

Other tutorials suggest storing sensitive information as environment variables to be accessed by your app on run time. Here we will be using a simpler file-based approach.

Start by creating a `config/database.yml` file:

```
deploy@server:~/sample_app$ nano config/database.yml
```

```
production:
  adapter: postgresql
  database: sample_app
  username: deploy
  encoding: unicode
  pool: 5
```

Note the lack of password thanks to peer authentication.

Also note that the code has a `config/database.yml.example` file and the `/config/database.yml` entry in `.gitignore`. In development, the developer can just copy over the example template to `config/-database.yml` and tweak its settings all the while being ignored by git on later commits.

Apart from database credentials, the only other production-specific settings we need to set in our simple app is the session secret. First generate a secret key using the `rake secret` task:

```
deploy@server:~/sample_app$ RAILS_ENV=production bin/rake secret
```

(We prefixed the environment variable `RAILS_ENV=production` to tell Rails to run the task in production. Otherwise, the task will run it in development and will throw an error due to missing development gems.)

Take note of the generated random key then overwrite the `secrets.yml` file:

```
deploy@server:~/sample_app$ nano config/secrets.yml
```

Remove everything and replace it with:

```
production:
  secret_key_base: [rake secret output]
```

Setup Production Database and Precompile Assets

Now for the final steps in the deploy user side.

Create and migrate the database using `rake db:create` and `rake db:migrate` under production:

```
deploy@server:~/sample_app$ RAILS_ENV=production bin/rake db:create
deploy@server:~/sample_app$ RAILS_ENV=production bin/rake db:migrate
== 20150520151700 CreatePosts: migrating ======================================
-- create_table(:posts)
   -> 0.0241s
== 20150520151700 CreatePosts: migrated (0.0243s) =============================
```

Then precompile the assets using `rake assets:precompile`:

```
deploy@server:~/sample_app$ RAILS_ENV=production bin/rake assets:precompile
```

As mentioned in the previous chapter, this (Sprockets / Asset Pipeline) is the only reason why we installed NodeJS. If you don't want to install NodeJS on your server, you can always add the `therubyracer` gem to your Gemfile. We will come back to this in a bit.

nginx/Passenger Settings

With the app fully set up, it's time to change our nginx settings to serve the new app.

Create a new server configuration file:

```
root@server:~# nano /etc/nginx/sites-available/rails
```

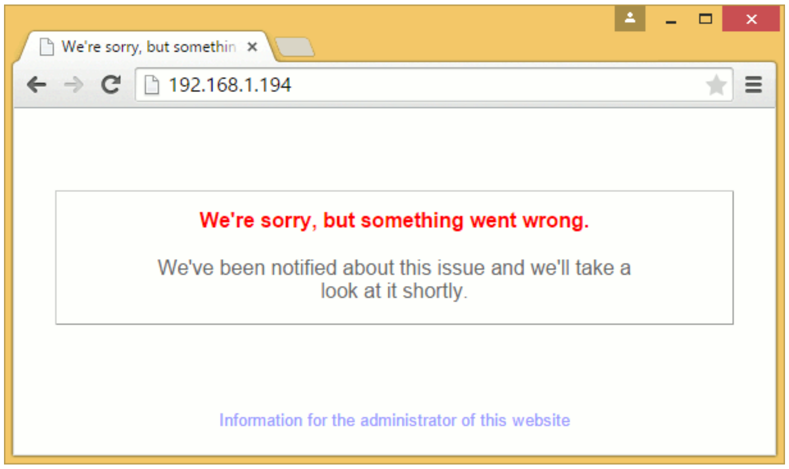
```
server {
        listen 80 default_server;
        listen [::]:80 default_server;
        server_name _;

        root /home/deploy/sample_app/public;
        passenger_enabled on;
}
```

Yes, it's almost the same as our Sinatra app's settings. This is because there's really no difference from the two apps: both are Rack applications with `public` as the static file folder.

Next, replace the Sinatra entry from `sites-enabled` with our new Rails entry and restart the server.

```
root@server:~# cd /etc/nginx/
root@server:/etc/nginx# rm sites-enabled/sinatra
root@server:/etc/nginx# ln -s ../sites-available/rails sites-enabled/rails
root@server:/etc/nginx# service nginx restart
```

Open your browser and you should now see…



We're sorry, but something went wrong.

We've been notified about this issue and we'll take a look at it shortly.

Information for the administrator of this website

…an error.

What happened here? Let's find out by opening the logs:

```
root@server:/etc/nginx# nano /var/log/nginx/error.log
```

Scroll up a bit to see the offending line:

```
[ 2015-06-03 21:00:05.9892 20894/7fbd7aab9700 App/Implementation.cpp:287 ]: Could not spawn process \
for application /home/deploy/sample_app: An error occured while starting up the preloader.
  Error ID: 3a7a406e
  Error details saved to: /tmp/passenger-error-eOEdZA.html
  Message from application: Could not find a JavaScript runtime. See https://github.com/rails/execjs\
 for a list of available runtimes. (ExecJS::RuntimeUnavailable)
  /home/deploy/sample_app/vendor/bundle/ruby/2.2.0/gems/execjs-2.5.2/lib/execjs/runtimes.rb:48:in `a\
utodetect'
```

It couldn't find a JavaScript runtime. Which is weird since we installed NodeJS system-wide.

Thankfully the solution is simple.

```
root@server:/etc/nginx# nano nginx.conf
```

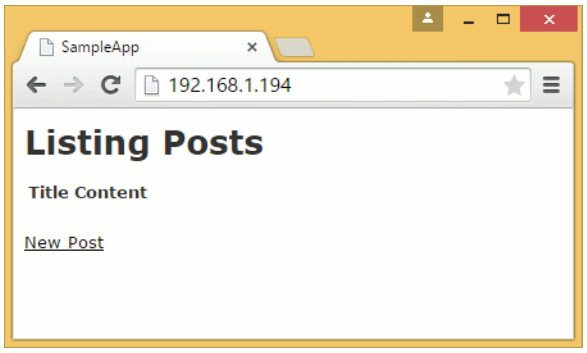
```
env PATH;
user www-data;
worker_processes 4;
```

By default, nginx removes almost all environment variables and this prevented ExecJS from finding the location of the NodeJS executable. Adding the `env` directive with the `PATH` environment variable allowed ExecJS to see the executable at `/usr/bin/nginx`.

Restart the server after adding the `env` line.

```
root@server:/etc/nginx# service nginx restart
```

You can now access the app in the browser.



Listing Posts

Title Content

New Post

On “`therubyracer`” gem

We would not have encountered the previous error if we went with the `therubyracer` gem; why don’t we just use that instead of NodeJS?

Historically, the problem with the `therubyracer` gem was its instability when installing and running it. But nowadays the main reason to avoid the gem is its large memory footprint.

Because of this, we will be using NodeJS as our JavaScript runtime for the entirety of this book.

Redeploying Rails Apps on Passenger

We've discussed how redeploying Sinatra apps only need a `touch tmp/restart.txt`. As you might expect, redeploying Rails apps can be a lot more complicated than that.

Let's look at a few examples of manually updating and restarting a Rails application. We'll see how we can simplify this process later.

Modifying the Rails App

The basic workflow for changing apps in production would be to push the changes to your repo then you would pull the changes on your server. Problem is, trying to simulate `git pull` in our practice app will require us to do slightly complicated `git reset` commands.

Instead of going through all that, we'll just use branches to simulate pulling changes from the origin repository:

```
deploy@server:~/sample_app$ git checkout bootstrap
M       config/secrets.yml
Branch bootstrap set up to track remote branch bootstrap from origin.
Switched to a new branch 'bootstrap'
```

This `bootstrap` branch changes the UI from the basic scaffold CSS to Bootstrap. This relatively large change means that we need to do a couple of things before we can restart the server.

First is to install the gems that provides the Bootstrap assets:

```
deploy@server:~/sample_app$ bundle install
```

Note that we didn't need to use the full command for `bundle install`. Those options were already saved under the `.bundle/config` file:

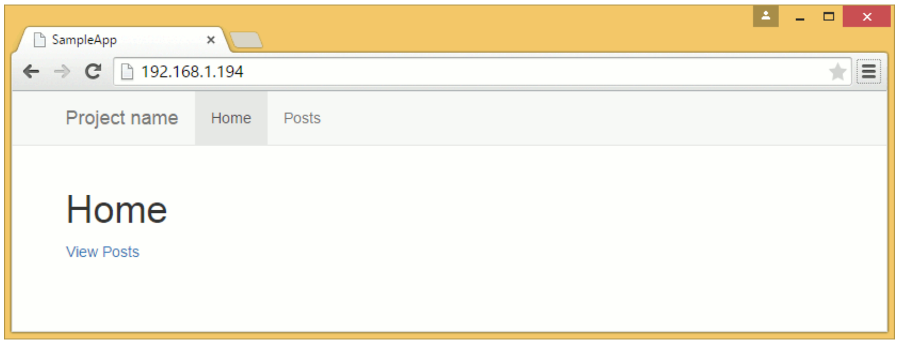
```
deploy@server:~/sample_app$ cat .bundle/config
---
BUNDLE_FROZEN: '1'
BUNDLE_PATH: vendor/bundle
BUNDLE_WITHOUT: development:test
BUNDLE_DISABLE_SHARED_GEMS: '1'
```

After installing all of the news gems, the next step is to precompile the new assets:

```
deploy@server:~/sample_app$ RAILS_ENV=production bin/rake assets:precompile
```

We can now restart our app and see its new UI.

```
deploy@server:~/sample_app$ touch tmp/restart.txt
```



Adding File Upload with Carrierwave

Sometimes we have to deploy new builds that require more than just a Bundler update. Let's try one such example:

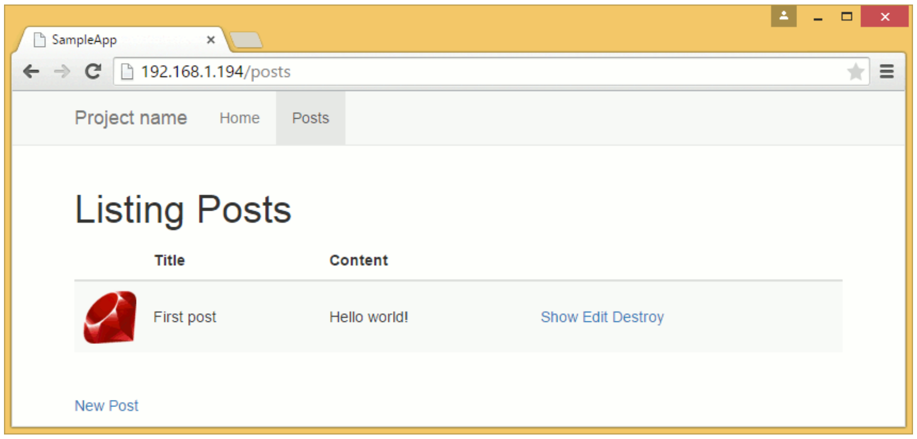
```
deploy@server:~/sample_app$ git checkout carrierwave
```

In this new build, we added file uploading through Carrierwave with thumbnail generation via the RMagick gem. These require both ImageMagick and libMagick so we switch back to root and install them before we continue with the deploy.

```
root@server:~# apt-get install imagemagick libmagickwand-dev -y
```

After the 150MB+ download and installation, we go back to our deploy user and run the commands for updating gems, migrating the database, precompiling assets, and restarting the application.

```
deploy@server:~/sample_app$ bundle

...

deploy@server:~/sample_app$ RAILS_ENV=production bin/rake db:migrate

...

deploy@server:~/sample_app$ RAILS_ENV=production bin/rake assets:precompile

...

deploy@server:~/sample_app$ touch tmp/restart.txt
```



Listing Posts

Title	Content	
First post	Hello world!	Show Edit Destroy

Simplifying Deployment with Mina

As you can see, most of the process in deploying new builds are just repetitive tasks. There's no reason we can't make scripts to automate the process.

Thankfully, there are existing tools that already do this. Capistrano is arguably the more popular deployment tool for Ruby, but we'll be covering Mina first. It has less features than the former, but for single machine deploys, it's much simpler (and sometimes even faster).

We'll be using a different project to explore Mina. Fork this repository to the online repository host of your choice then check out the fork on your local machine.

```
$ git clone git@github.com:[username]/sample_mina_app.git
```

You might notice in the project's `Gemfile` that this application uses SQLite as its development and test database and PostgreSQL in production. This is intentional: we expect that some of you will try to deploy your own apps that were started off using SQLite and doing the same in the sample app will demonstrate that the deployment process will work just fine even when using the said scheme.

We strongly suggest that you use the same database for your development and production environments for your future projects, though.

Initial Setup

Let's install the gem and generate the default `deploy.rb` settings file.

```
$ cd sample_mina_app
$ bundle install
$ mina init
-----> Created ./config/deploy.rb
       Edit this file, then run `mina setup` after.
```

We will be deploying to `/home/deploy/sample_mina_app`. Open `config/deploy.rb` and change the following lines to your practice server settings:

```
~~set :domain, 'foobar.com'~~
~~set :deploy_to, '/var/www/foobar.com'~~
~~set :repository, 'git://...'~~
~~set :branch, 'master'~~
set :domain, '[ip address]'
set :deploy_to, '/home/deploy/sample_mina_app'
set :repository, 'https://github.com/[username]/sample_mina_app.git'
set :branch, 'master'
set :user, 'deploy'
```

Settings for NAT networked servers are slightly different:

```
set :domain, 'localhost'
set :deploy_to, '/home/deploy/sample_mina_app'
set :repository, 'https://github.com/[username]/sample_mina_app.git'
set :branch, 'master'
set :user, 'deploy'
set :port, '2222'
```

The repository setting above uses HTTPS instead of SSH to pulling of code without having credentials. We'll discuss later how to use deploy keys to allow checking out of code via SSH.

Mina Directory Structure

Mina creates the following directory structure for projects:

```
/home/deploy/sample_mina_app/   # The deploy_to path
 |-  releases/                  # Holds releases, one subdir per release
 |   |- 1/
 |   |- 2/
 |   |- 3/
 |   '- ...
 |-  shared/                    # Holds files shared between releases
 |   |- logs/                   # Log files are usually stored here
 |   `- ...
 '-  current/                   # A symlink to the current release in releases/
```

Everytime a new build is deployed, Mina checks it out to a new folder under `releases`. It then symlinks files and folders from the `shared` folder before running the other tasks (e.g. `bundle install`). If the deploy is successful, the `current` folder is symlinked to the current release.

Clean builds are checked out to the `release` folder so our previous strategy for `secrets.yml` (simply overwiting it) will not work. Instead, we will have to put the `secrets.yml` and `database.yml` under the shared folder and let Mina symlink them at the symlinking step.

Modify the `set :shared_paths` line to add the `secrets.yml`:

```
~~set :shared_paths, ['config/database.yml', 'log']~~
set :shared_paths, ['config/database.yml', 'config/secrets.yml', 'log']
```

We also need to modify the Mina setup script to give us empty files to configure later.

```
task :setup => :environment do
  queue! %[mkdir -p "#{deploy_to}/#{shared_path}/log"]
  queue! %[chmod g+rx,u+rwx "#{deploy_to}/#{shared_path}/log"]

  queue! %[mkdir -p "#{deploy_to}/#{shared_path}/config"]
  queue! %[chmod g+rx,u+rwx "#{deploy_to}/#{shared_path}/config"]

  queue! %[touch "#{deploy_to}/#{shared_path}/config/database.yml"]
  queue  %[echo "-----> Be sure to edit '#{deploy_to}/#{shared_path}/config/database.yml'."]

  queue! %[touch "#{deploy_to}/#{shared_path}/config/secrets.yml"]
  queue  %[echo "-----> Be sure to edit '#{deploy_to}/#{shared_path}/config/secrets.yml'."]
end
```

Note that the `log` folder is also symlinked and shared between releases. Otherwise, the production logs will be lost everytime you deploy a new build.

Other folders like `vendor/bundle` and `public/assets` are also shared between builds but they are handled by different tasks and don't need to be set in `:shared_paths`.

Carrierwave Files

File uploading libraries like Carrierwave are often placed under the `public` folder. Like logs, they will be lost if you do not add them to the `shared` files.

For Carrierwave, you'll need to add the `public/uploads` folder to the Mina settings i.e.

```
set :shared_paths, ['config/database.yml', 'config/secrets.yml', 'log',
  'public/uploads']

...


task :setup => :environment do
  queue! %[mkdir -p "#{deploy_to}/#{shared_path}/public/uploads"]
  queue! %[chmod g+rx,u+rwx "#{deploy_to}/#{shared_path}/public/uploads"]
```

Server Setup

We should now be ready to deploy with Mina. But first we need to let Mina create the directoy structure as well as the files we configured in the setup task.

(**Windows users**, please see the note at the end of this chapter before continuing.)

```
$ mina setup
...
-----> Be sure to edit '/home/deploy/sample_mina_app/shared/config/database.yml'.
-----> Be sure to edit '/home/deploy/sample_mina_app/shared/config/secrets.yml'.
```

Switch over to the `deploy` SSH session and edit the database config file.

```
deploy@server:~$ nano sample_mina_app/shared/config/database.yml
```

```
production:
  adapter: postgresql
  database: sample_mina_app
  username: deploy
  encoding: unicode
  pool: 5
```

Then the session secret (generate one locally by running `rake secret`):

```
deploy@server:~$ nano sample_mina_app/shared/config/secrets.yml
```

```
production:
  secret_key_base: [rake secret output]
```

Then create the database:

```
deploy@server:~$ createdb sample_mina_app
```

Configuring a new nginx server entry and restarting nginx to apply the changes are left as exercises to the reader. (Hint: the root should be “`current/public`”.)

Deploying

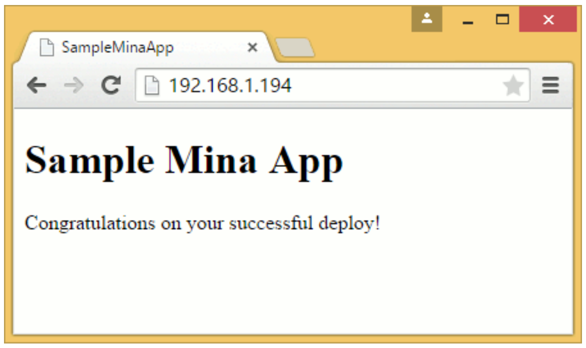
Now that everything is ready, we can now deploy the latest build through Mina:

```
$ mina deploy
-----> Creating a temporary build path
-----> Cloning the Git repository
       Cloning into bare repository '/home/deploy/sample_mina_app/scm'...
-----> Using git branch 'master'
       Cloning into '.'...
       done.
-----> Using this git commit
...
```

Mina will automatically perform all of the manual deployment steps that we did in the previous section. The "`deploy`" task in `config/deploy.rb` lists down all of those steps.

```
desc "Deploys the current version to the server."
task :deploy => :environment do
  to :before_hook do
    # Put things to run locally before ssh
  end
  deploy do
    # Put things that will set up an empty directory into a fully set-up
    # instance of your project.
    invoke :'git:clone'
    invoke :'deploy:link_shared_paths'
    invoke :'bundle:install'
    invoke :'rails:db_migrate'
    invoke :'rails:assets_precompile'
    invoke :'deploy:cleanup'

    to :launch do
      queue "mkdir -p #{deploy_to}/#{current_path}/tmp/"
      queue "touch #{deploy_to}/#{current_path}/tmp/restart.txt"
    end
  end
end
```

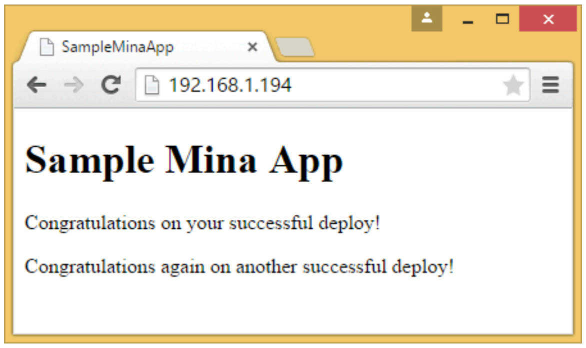
It pulls a new build, links the shared files, runs `bundle install`, etc. It even assumes that we are using Passenger and includes `touch tmp/restart.txt` by default to restart the application.



Sample Mina App

Congratulations on your successful deploy!

Try to redeploy a new build: modify a file in the project (`/app/views/pages/home.html.erb` is a good candidate), commit and push it to your repository, then run "`mina deploy`" again. Refresh the browser to see the changes.



Sample Mina App

Congratulations on your successful deploy!

Congratulations again on another successful deploy!

As you can see, having a deployment tool like Mina significantly reduces the amount of effort needed to deploy new builds.

Mina on Windows

To put it bluntly, Mina has a bug preventing it from running on Windows

As a Windows user, this would be a great time for you to to start switching over to using Linux as your development environment. Windows was fine when you were still learning the ropes, but now that you are going deeper into Rails development, you should seriously consider matching your dev environment to your target production environment. That means using plain VMs under VirtualBox/VMware or using tools like Vagrant. (Yes, many Mac users also do this - OS X *is* in fact different from Linux.)

If for some reason you absolutely have to run Mina on Windows, you can follow the workaround provided in the bug report. First find out where the Mina gem was installed via:

```
$ bundle show mina
```

The path could be "`c:/RailsInstaller/Ruby2.1.0/lib/ruby/gems/2.1.0/gems/mina-0.3.4`"; the file you're looking for is "`lib/mina.rb`" under that folder. Open it with your text editor and add the following lines after the module declaration:

```
module Mina
 #if windows os
 require 'rbconfig'
 is_windows = (RbConfig::CONFIG['host_os'] =~ /mswin|mingw|cygwin/)
 if is_windows
   module Shellwords
     def shellescape(str)
         '"' + str.gsub(/\\(?=\\*\")/, "\\\\\\").gsub(/\"/, "\\\"").gsub(/\\$/, "\\\\\\").gsub("%", \
"%%") + '"'
     end

     module_function :shellescape
     class << self
```

```
        alias escape shellescape
      end
    end
  end
...
```

Mina should now work under Git Bash. (It won’t work under “Start Command Prompt with Ruby on Rails” because that terminal lacks `ssh`.)

Deploying on a VPS

You should now be ready to deploy your Ruby application on a real VPS.

Overview of Popular VPS Providers

There are may VPS providers out there. To help you decide what to choose, we'll look at three of the more popular providers: Linode, DigitalOcean, and Amazon.

Linode

We start with Linode because it's a safe bet for many developers: its VPSs are affordable, it has data centers around the world, and it has a fairly good reputation for the past decade.

Pricing and Billing

Linode's lowest priced server is at $0.015 an hour or $10 for a whole month. This will give you a virtualized server with 1GB RAM, 1 CPU core, and 24 GB of SSD storage, more than enough for serving a small to medium-scale web application.

Linode requires a valid credit card to setup servers, and will be billed at the end of the month automatically. There is a 7-day money back guarantee for new users if you wish to cancel your account early.

Setting Up a Server in Linode

Linode's Getting Started guide is a good step-by-step walkthrough on setting up your new Linode server. To sync with this book, however, you should use Ubuntu 14.04 LTS instead of the guide's suggested 12.04 LTS, and you should prefer setting the timezone to UTC; we can always change the displayed time zone under Rails later.

You might have noticed that Linode immediately gives you a root user. To avoid shooting yourself in the foot, create a new admin user with `sudo` privileges (i.e. the `user` user in the past few chapters) with the help of the Securing your Server guide. After creating the admin user, you should now be able to continue the steps provided under the *Additional Server Setup* section. (You should also do the other two steps in the guide, creating a firewall and installing `fail2ban`, since your server is now publicly available on the internet.)

And that's basically it for Linode - just follow the steps we did in the previous chapters to deploy your own Rails app on the server. If you're just planning to use this as a test, don't forget to destroy the instance as merely shutting down the server will not stop the billing of the instance.

Digital Ocean

Another popular VPS provider is DigitalOcean. A relative newcomer, DO was one of the reasons why VPSs are cheap nowadays. Back in the day when many VPSs were priced at $20 a month, DO came out with $5 VPSs with SSD storage, something unheard of in the low-end server space.

Even though the market has somewhat caught up, DO is still a viable choice for VPS hosting.

Pricing and Billing

DigitalOcean's $5 server is still available and at 512MB RAM, it should be able to serve small-scale or practice applications. You should go for the higher tiers if you're deploying applications that have a decent amount of traffic.

Like Linode, requires a valid credit card before you can create a server. One thing DO has that many VPS providers don't is a PayPal payment option, though this requires you to pre-load credit from PayPal to your account or face monthly Termination Notices.

Instead of a 1 week money-back guarantee, DO has promo codes that give you credit when you sign up. For example, using `DROPLET10` will give you $10 of credit. Combined with a valid credit card, this can give the new user 1-2 months of free use (a verification fee of $1.23 is charged to the card but is quickly refunded upon verification).

Setting Up a Server in DigitalOcean

DO has its own Getting Started guide, with the walkthrough at How To Create Your First DigitalOcean Droplet Virtual Server. As with Linode, you should use Ubuntu 14.04 to match the OS used in this book.

DO lets you provide a public key before spinning up the instance of your server. If you do this, DO will perform all of the steps to adding the key (e.g. creating the `.ssh/authorized_keys`) allowing you to immediately login as `root` remotely. As with Linode, it's recommended to create a separate admin user and prevent remote `root` login. This guide explains the process; it's similar to Linode's guide, with the only difference being the command used - `gpasswd` instead of `usermod`. The two practically do the same thing so it comes to personal preference (I personally prefer `gpasswd`).

Configuring the timezone and firewall is detailed in another document. They are somewhat important, but not as important as the section on Creating a Swap File.

Unlike other VPSs, DO "droplets" **do not** have swap files (where Linux stores its Virtual Memory) configured by default and you should create them manually. So before you proceed with trying to deploy your Rails app on your DO droplet, you must create a swap file which is at least as large as your droplet's memory (e.g. `sudo fallocate -l 512M /swapfile` or more for 512MB droplets, `1G` or more for 1GB droplets, etc).

Amazon Elastic Compute Cloud (EC2)

An alternative to traditional VPSs would be to use a cloud computing platform to host your entire system. Amazon Web Services (AWS) is a well-known example; not only does it provide virtual machines via Amazon EC2, it also provides online file storage via Amazon S3, content delivery via Amazon CloudFront, scalable databases via Amazon RDS, and so on.

Here we'll look at Amazon EC2, the VPS equivalent in AWS.

Pricing and Billing

Being part of a cloud platform, Amazon EC2 is billed hourly and thus requires a valid credit card. The pricing is also somewhat complicated, with price differences not only on tiers and types of instances, but also on the location of the data center.

Per dollar, Amazon EC2 instances are more expensive than VPSs with similar specs and performance from other providers. For example, the lowest tier, `t2.micro` is about as twice as expensive as DO's $5 tier but can sometimes be slower than the latter. It may not be financially sound for a beginner to use Amazon EC2 for their small to medium-scale applications, especially if their application doesn't use the other parts of AWS.

On the plus side, AWS provides a free tier for new sign-ups. This lets you use, among other things, a `t2.micro` instance for 12 months, far longer than the free options from both Linode and DO.

Setting Up a Server in Amazon EC2

Amazon EC2 also has a Getting Started Guide as well as a Setting Up guide for steps to perform before launching a new instance.

Connecting to the instance is slightly different as you cannot remotely log in using passwords or your own public key. Instead, you have to let AWS generate the private-public key pair by following the **Create a Key Pair** section of *Setting Up* guide and download the `.pem` file that you will use to log in later. You should also follow the rest of the *Setting Up* guide especially the **Create a Security Group** section which will open the SSH port for remote access.

For practice, choose to create a `t2.micro` instance running Ubuntu 14.04 LTS and use the newly created "existing" key. After logging in as the `ubuntu` (**not** `ec2-user` which is Amazon Linux's default user), you can now continue following the steps in the previous chapters to set up your Rails server. You can also copy over your personal public key on a new line at the end of `.ssh/authorized_keys` if you wish to connect using your own private key.

Note that EC2 instances don't have swap files by default due to the nature of the default storage (EBS). You can create a swap file by following the steps in DO's guide, but you should probably not do this since I/O in EBS is billed. If you're hitting the memory limit, it's better to get higher tiered instances with more memory or tiers with Instance Store Volumes where you can put a swap file without being charged for I/O.

Setting Up Your Custom Domain

coming soon