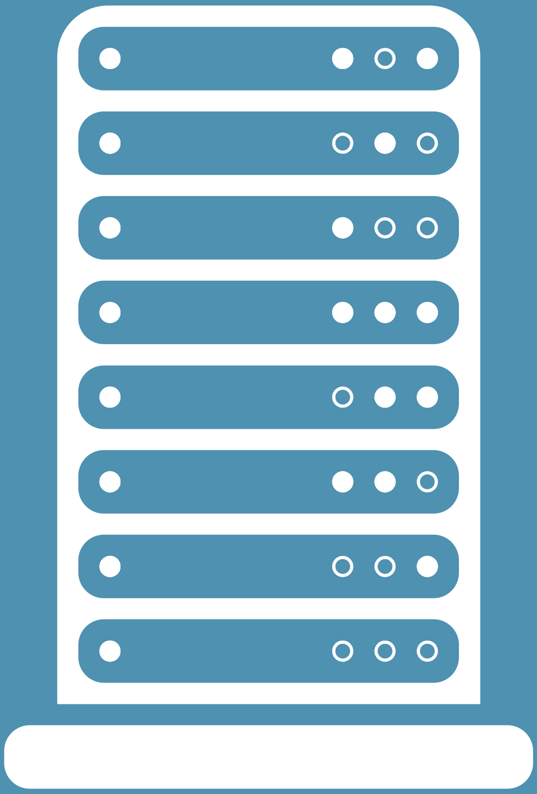
Michael Trojanek

Build Your Own Rails Server

A step-by-step guide for Rails developers.



Second edition v2

Table of contents

1 Welcome

Thank you very much for taking the time to read this book!

Whether you are a Rails developer sick of opinionated blackbox-tools for easy-as-pie deployment (that only work if you adjust your environment to their requirements), want to know how a server running a Rails application handles requests or just want to augment your developing skills by learning how the Rails stack works from an operator's point of view – I am sure this book will be of great value for you.

When you are through, you will have a virtual linux box serving a sample Rails application built completely from scratch and you will know which steps are necessary to deploy a new release.

The goal of this book is neither pasting a list of commands into your terminal nor building an automated deployment stack – it is making sure that not only you know *how* to carry out each step, but to understand *why* each component is configured the way it is.

In case you get stuck for whatever reason, do not hesitate to write an email to michael.trojanek@relativkreativ.at or ping @relativkreativ on Twitter – I'll glad to help.

While this book is free, it is intended as a giveaway for people who join my email list. If you came across it without joining – no problem (I even encourage you to share it among friends and co-workers). Nonetheless, a lot of effort went into writing it and it would be great if you joined my list as a sign of respect (you can do that here). Not only will you get free updates but I bet my emails will have something in store for you.

1.1 Conventions

There is not much to be said about the style conventions used in this book – you just need to pay attention to two important things:

- **The command prompt**
  I try to be clear in my writing but the command prompt will ultimately tell you whether a command has to be run on your development machine or on our server. To make it easy to distinguish, my development machine is called `development` and my user is `michael` (so `[michael@development ~]$` means you on your local machine, `[root@server ~]#` means the root user on our server and `[app@server ~]$` means the app user on our server).
- **The line numbers of file listings**
  Some lines in configuration files may be quite long and spawn multiple lines in this book. Be careful not to introduce wrong line breaks – missing line numbers on the left hand side indicate that a directive spawning multiple lines in the book has to be written as one single line in the real file.

Commands you have to run as well as additions to longer files are set in bold.

1.2 Preparation

We will use Oracle VirtualBox as our virtualization tool of choice. While it is lacking performance compared to other virtualization tools, it's free and it runs on all platforms. If you are not already using it, download it from here (just go with the platform package suiting your platform, we do not need the Expansion Pack) and install it.

While we could configure a virtual machine completely from scratch, we will take a different approach and instead of messing around with defining virtual networks and configuring virtual harddisk controllers, we will concentrate on bringing a Linux installation from bare to Rails hosting (if you rent a VPS, it comes preinstalled with an operating system anyway so i want to keep the step of provisioning the machine as short and painless as possible).

For this purpose, we will use Vagrant (a tool actually intended to manage virtual development servers) to download and reuse a preconfigured virtual machine (thus skipping the configuration step) and tweak it a little.

Get Vagrant's installer from its download page and install it.

Vagrant uses predefined virtual machines (called *boxes*) you can configure to your needs with a configuration file. The first time you make use of a box, it has to be downloaded to your machine (which can take some time depending on your network connection since it is a complete VM weighting a few hundred MBs).

We will use a minimal CentOS 7 image I built myself as a starting point. So, as our last step of preparation, download the box with the following command:

```
[michael@development ~]$ vagrant box add relativkreativ/centos-7-minimal
==> box: Loading metadata for box 'relativkreativ/centos-7-minimal'
    box: URL: https://atlas.hashicorp.com/relativkreativ/centos-7-minimal
==> box: Adding box 'relativkreativ/centos-7-minimal' (v1.0.4) for provider:
virtualbox
    box: Downloading: https://atlas.hashicorp.com/relativkreativ/boxes/centos-7-
minimal/versions/1.0.4/providers/virtualbox.box
==> box: Successfully added box 'relativkreativ/centos-7-minimal' (v1.0.4) for
'virtualbox'!
[michael@development ~]$
```

In case you want to build your own Vagrant base box, I published a detailed tutorial on how to do that on my website.

2 Preparation

By the end of this chapter, we will have created a virtual machine with a bare operating system. Its network interface will be configured as a bridged adapter which means that this virtual machine will act as is it was a physical box connected to our network.

In addition, we will locally map the server's IP address to a name.

Since you already installed VirtualBox and Vagrant, we can dive right in and prepare a fresh virtual machine.

2.1 Define the virtual machine

Create a directory somewhere on your harddrive. I will create mine in my home folder and name it `byors` (for **B**uild **Y**our **O**wn **R**ails **S**erver in case you're wondering). If you want the commands from this book to be copy-and-pasteable, I suggest you do the same or link the directory to your home folder if you rather want to create it somewhere else.

```
[michael@development ~]$ mkdir byors
[michael@development ~]$ cd byors
[michael@development byors]$
```

Now we initialize Vagrant which will create a Vagrantfile holding our server's configuration:

```
[michael@development byors]$ vagrant init
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
[michael@development byors]$
```

Edit this file so Vagrant uses the box we just downloaded as a strating point. The default Vagrantfile is quite big (consisting mostly of comments listing the available configuration directives) but since we do not have any special requirements, simply replace the file's contents with the following lines:

~/byors/Vagrantfile

```
Vagrant.configure(2) do |config|
  config.vm.box = "relativkreativ/centos-7-minimal"
end
```

2.2 Bring it up

We make Vagrant provision and boot the machine by running the command `vagrant up`:

```
[michael@development byors]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'relativkreativ/centos-7-minimal'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'relativkreativ/centos-7-minimal' is up to date...
==> default: Setting the name of the VM: byors_default_1457978505987_67207
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 (guest) => 2201 (host) (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant => /Users/michael/byors
[michael@development byors]$
```

Take a look at the line which reads `default: SSH address: 127.0.0.1:2222`.

Vagrant is creating the virtual machine with a NAT'ed network adapter (VirtualBox' default) which means that while the VM will be able to access the internet (by using our development machine's internet connection) our server will not be directly accessible from the outside (which even means our development machine where the box is hosted).

So right now, if we wanted to login to our server via SSH, we have to either connect to port 2222 on our development machine (`ssh 127.0.0.1 -p 2222`) or use Vagrant's built-in command (`vagrant ssh`).

If a Rails application would run on our server, we had no way of requesting it with a browser at the moment which is why we will add a second network interface in the next step.

2.3 Add a network interface

A real production server uses a fixed public IP address which is accessible from all over the internet. We will mimic this behaviour by adding a bridged network interface to our virtual machine. This way our server will be accessible from within our private network, but not from the outside. And more importantly, it will behave like a real production machine.

Unfortunately, in order to configure a static IP address, we need to know what private network class we are using and it is impossible for me to tell which one you are using. There is not even a common default – if you have not configured anything (or if you simply do not know what I am talking about) then you are using whatever you router's manufacturer defined as default.

If you are not sure what network class you are using, go to your development machine's network settings and look for an IP address starting with `10`, `192` or `172`. Alternatively you can use the terminal command `ifconfig` on Unix-based systems to list all active network adapters and their configuration.

Networking is a complex thing, but as rule of thumb take your development machine's IP address and change its last segment to a number less than 255 – my development machine uses the IP address `192.168.0.5` so I will use `192.168.0.100` for our server.

It is easy to tell Vagrant to add an interface. Edit the `Vagrantfile` and add the following line:

~/byors/Vagrantfile

```
Vagrant.configure(2) do |config|
  config.vm.box = "relativkreativ/centos-7-minimal"
  config.vm.network "public_network", ip: "192.168.0.100"
end
```

This will make Vagrant configure an additional network interface for the VM and assign it the fixed IP address `192.168.0.100`.

Since your development machine probably has more than one interface, Vagrant will ask you which interface you want the VM's interface to attach to as soon as you apply this `Vagrantfile` or rebuild the server next time.

If you do not want to manually select the interface each time you rebuild the machine, remember the exact name of your choice and specify it in the Vagrantfile. To get a list of our choices, let's first reload our Vagrant configuration:

```
[michael@development byors]$ vagrant reload
==> default: Attempting graceful shutdown of VM...
==> default: Checking if box 'relativkreativ/centos-7-minimal' is up to date...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Available bridged network interfaces:
1) en1: WLAN (AirPort)
2) en2: Thunderbolt 1
3) en3: Thunderbolt 2
4) p2p0
5) awdl0
6) en0: Ethernet
7) bridge0
==> default: When choosing an interface, it is usually the one that is
==> default: being used to connect to the internet.
    default: Which interface should the network bridge to? 1
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: bridged
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /vagrant => /Users/michael/byors
==> default: Machine already provisioned. Run `vagrant provision` or use the `--
provision`
==> default: flag to force provisioning. Provisioners marked to run always will
still run.
[michael@development byors]$
```

As you can see, I am connecting to my network through my Mac's Airport interface, so this is the name I use:

~/byors/Vagrantfile

```
Vagrant.configure(2) do |config|
  config.vm.box = "relativkreativ/centos-7-minimal"
  config.vm.network "public_network", ip: "192.168.0.100" , bridge: "en1: WLAN (AirPort)"
end
```

Strictly speaking, this is not exactly how a production machine would be configured (since a production server from a hosting provider typically has just one network interface). However, we can simply ignore the first interface when we build our server – completely removing it is a complex task and would break most of Vagrant's functionality.

You can verify that everything works correctly by pinging our server's IP address and see if it responds:

```
[michael@development byors]$ ping -c 4 192.168.0.100
PING 192.168.0.100 (192.168.0.100): 56 data bytes
64 bytes from 192.168.0.100: icmp_seq=0 ttl=64 time=0.401 ms
64 bytes from 192.168.0.100: icmp_seq=1 ttl=64 time=0.218 ms
64 bytes from 192.168.0.100: icmp_seq=2 ttl=64 time=0.242 ms
64 bytes from 192.168.0.100: icmp_seq=3 ttl=64 time=0.211 ms

--- 192.168.0.100 ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.211/0.268/0.401/0.078 ms
[michael@development byors]$
```

2.4 Add a hosts entry

Right now, we have to remember our server's IP address – to access a website or application running on our server, we would have to type `http://192.168.0.100/some/url` in our browser's address bar.

Not very convenient (after all, this is why DNS has been invented).

Instead of adding a DNS entry (which would not work in this case because our server is not publicly accessible from the internet like our production machine would be) we will add a local hosts entry which maps the server's IP to a name which is easy to remember.

Edit the file `/etc/hosts` (for Unix-based systems) or `C:\Windows\System32\drivers\hosts` (if you work on Windows) on your development machine and add the following line (you probably need to use `sudo` or become admin to make changes to this file):

/etc/hosts or C:\Windows\System32\drivers\hosts

```
192.168.0.100 byors.local www.byors.local
```

This maps `byors.local` and `www.byors.local` to the IP address `192.168.0.100` so from now on we can request a website or application running on our server by typing `http://byors.local` or `http://www.byors.local` in a browser's address bar.

Even though we did nothing Rails-specific so far, these tasks were important to prepare a common starting point. Our new server is now configured more or less exactly like fresh VPS you would rent from a hosting company.

2.5 Take a snapshot

At this point, we could use the command `vagrant destroy` to destroy our virtual machine, bring it back up with `vagrant up` and it would start humming along as if nothing happened.

Nonetheless I encourage you to take a snapshot of the machine's current state so you can return to this point in time in case something goes wrong in one of the following chapters.

Even though you can use VirtualBox' GUI to take a snapshot of your machine (it will be named something like `byors_default_` followed by a long string of numbers) it is easier to do that from within Vagrant. The command `vagrant snapshot save` will create a snapshot which you can later restore with `vagrant snapshot restore`:

```
[michael@development byors]$ vagrant snapshot save "End of chapter 2"
==> default: Snapshotting the machine as 'End of chapter 2'...
==> default: Snapshot saved! You can restore the snapshot at any time by
==> default: using `vagrant snapshot restore`. You can delete it using
==> default: `vagrant snapshot delete`.
[michael@development byors]$
```

Now let's begin to turn this bare server into a Rails machine!

3 Provisioning

In this chapter we will first take a look at how a server running Rails handles incoming requests. Then we will install a webserver and a database system.

3.1 How our server handles requests

Let us first examine how a typical Rails request will be handled by our stack:

- The request comes in to the webserver on port 80 (or port 443 if you serve your application encrypted via SSL).
- The webserver's master process (nginx in our case) hands off the request to one of its workers.
- If the visitor's browser requested a static file (an image, a cached page, a compiled stylesheet or JavaScript file), the worker serves this file directly (circumventing the Rails stack).
- If it is a Rails request, the nginx worker hands off the request to the Unicorn applicaiton server (which waits for connections listening on a dedicated port).
- The Unicorn master process distributes incoming requests among its worker processes (each of these has a copy of our application in memory and holds a connection to the database).
- The requested page is generated and handed back to the visitor's browser.

We will now begin with installing our webserver and database. The Unicorn application server is quite dependent on our application, so we will take care of that in the next chapter.

3.2 Install the webserver

When it comes to webservers for Rails applications, you basically have the choice between Apache and nginx. While Apache is one of *the* OpenSource projects, nginx is quite popular in the Rails community. It is less bloated, extremely fast and needs a lot less resources compared to Apache.

In our configuration, the webserver has two main tasks:

- It acts as a proxy to our application server (accepting requests and forwarding them).
- It serves static assets (images, compiled stylesheets and JavaScript files).

nginx is not available in CentOS' default repository, so a simple `yum install` does not work. As the nginx installation instructions show, we either have to download the prepackaged .rpm-file from the nginx website or define the nginx repository and the install nginx with yum. We will choose the latter because it will make updating nginx in the future much easier.

This is the first time we log in as root on our machine. The default root password for virtual machines created with Vagrant is `vagrant`. We use this password when prompet after initiating a SSH session to our server:

```
[michael@development byors]$ ssh root@byors.local
The authenticity of host 'byors.local (192.168.0.100)' can't be established.
ECDSA key fingerprint is SHA256:RtRS8hPpQebDC7w5+SXMLygzuLhHMPiSyX2hDTJE41U.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'byors.local' (ECDSA) to the list of known hosts.
root@byors.local's password: vagrant
Last login: Tue Mar 15 13:55:57 2016
[root@server ~]#
```

Logged in as root, the first thing we do is creating a file called `nginx.repo` in the directory `/etc/yum.repos.d`. Create it with the following contents (taken directly from nginx' installation instructions):

/etc/yum.repos.d/nginx.repo

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/centos/$releasever/$basearch/
gpgcheck=0
enabled=1
```

Now we can install the nginx webserver with yum:

```
[root@server ~]# yum install nginx

...

================================================================================
 Package         Arch            Version                    Repository      Size
================================================================================
Installing:
 nginx           x86_64          1:1.8.1-1.el7.ngx          nginx          372 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 372 k
Installed size: 897 k
Is this ok [y/d/N]: y

...

Installed:
  nginx.x86_64 1:1.8.1-1.el7.ngx

Complete!
[root@server ~]#
```

nginx is currently not configured to start when our server boots which we can verify by grepping systemctl's unit files for `nginx.service`:

```
[root@server ~]# systemctl list-unit-files | grep nginx.service
nginx.service                                  disabled
[root@server ~]#
```

To make it start when our server boots, we use the following command:

```
[root@server ~]# systemctl enable nginx.service
ln -s '/usr/lib/systemd/system/nginx.service' '/etc/systemd/system/multi-
user.target.wants/nginx.service'
[root@server ~]#
```

What's left is starting our webserver:

```
[root@server ~]# systemctl start nginx.service
[root@server ~]#
```

3.2.1 Adjust the firewall

Even though our webserver is now installed and started, we will not be able to request its default welcome page with a browser. That's because our server's default firewall configuration currently blocks requests to port 80 – you can verify that by directing your browser to http://byors.local (which will timeout after a while).

In order to open port 80 on our firewall, we have to first modify our network interfaces' default zones. Since we are building a server, we will configure them to use the block zone as default (this zone rejects all requests).

First we take a look at all network interface configuration files for our machine:

```
[root@server ~]# ls /etc/sysconfig/network-scripts/ifcfg-*
/etc/sysconfig/network-scripts/ifcfg-enp0s3
/etc/sysconfig/network-scripts/ifcfg-enp0s8
/etc/sysconfig/network-scripts/ifcfg-lo
[root@server ~]#
```

`ifcfg-lo` represents the local loopback interface (`localhost` or `127.0.0.1`), the others represent our bridged interface and the one Vagrant adds by default (because of CentOS' naming scheme, they may have different names on your system but the local loopback interface is always called `lo`).

Both of these files have no default firewall zone configured so we have to add the following line to the end of each of these two files:

/etc/sysconfig/network-scripts/ifcfg-enp0s3 and /etc/sysconfig/network-scripts/ifcfg-en0ps8

```
ZONE=block
```

After applying our changes *every* connection to our network interfaces will be dropped (this includes SSH requests) so we have to be careful not to lock ourselves out. Before we make our network interfaces use their new default zones, we will allow SSH and HTTP connections in this zone:

```
[root@server ~]# firewall-cmd --zone=block --add-service=ssh --permanent
success
[root@server ~]# firewall-cmd --zone=block --add-service=http --permanent
success
[root@server ~]#
```

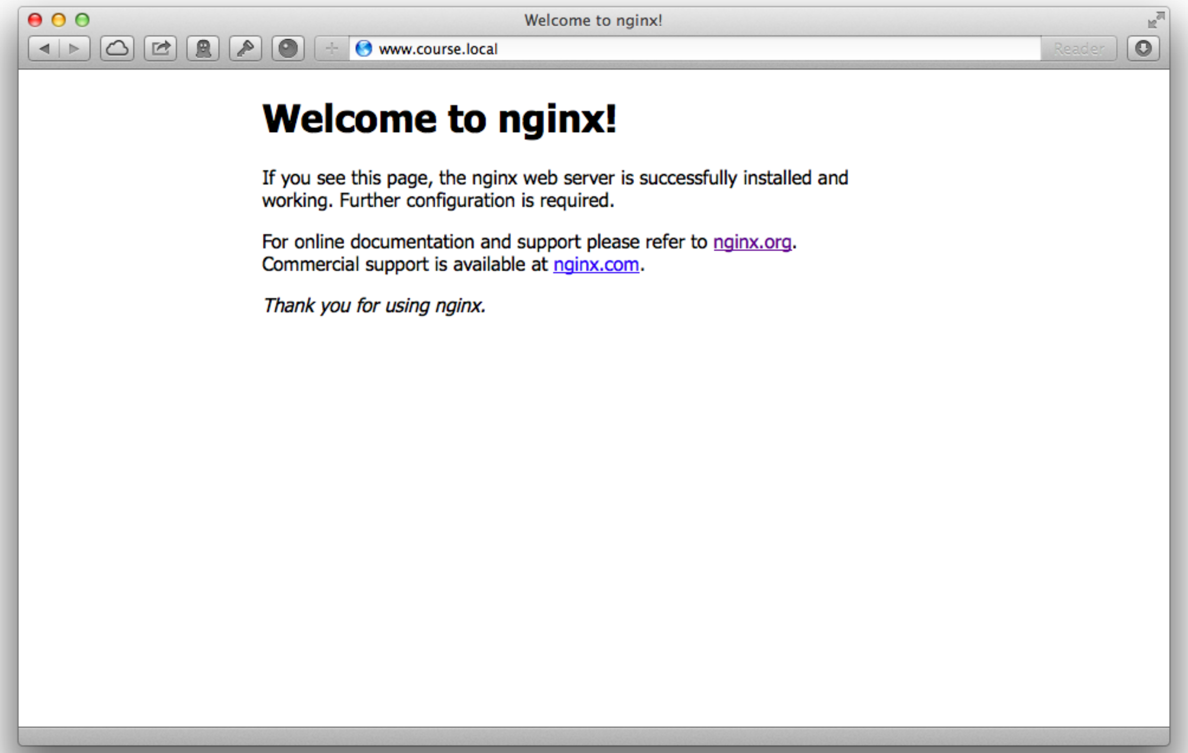
We can verify that everything worked as expected by listing all permanently allowed services for the block zone:

```
[root@server ~]# firewall-cmd --zone=block --list-services --permanent
http ssh
[root@server ~]#
```

Now that everything is configured correctly, we will reboot our server which will make our network interfaces come up with their new default zones:

```
[root@server ~]# reboot
Connection to byors.local closed by remote host.
[michael@development byors]#
```

When our server has rebooted, we can verify that our firewall is now configured to allow HTTP requests to port 80 by firing up a webbrowser and visiting the URL http://byors.local – we will be greeted by nginx' default welcome page:



Welcome to nginx!

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

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

3.3 Install the database

In development, you probably use SQLite because it is easy to manage and provides a perfect small-scale database system. However, you would not want to use it in production mainly because it is not transaction-safe and performance is not really among its strengths.

We are going to use MariaDB (which is CentOS 7's drop-in replacement for MySQL):

```
[michael@development byors]$ ssh root@byors.local
root@byors.local's password: vagrant
Last login: Wed Mar 16 00:42:17 2016 from 192.168.0.118
[root@server ~]# yum install mariadb-server

...

================================================================================
 Package                     Arch      Version                  Repository   Size
================================================================================
Installing:
 mariadb-server              x86_64    1:5.5.44-2.el7.centos    base         11 M
Installing for dependencies:
 mariadb                     x86_64    1:5.5.44-2.el7.centos    base        8.9 M
 perl-Compress-Raw-Bzip2     x86_64    2.061-3.el7              base         32 k
 perl-Compress-Raw-Zlib      x86_64    1:2.061-4.el7            base         57 k
 perl-DBD-MySQL              x86_64    4.023-5.el7              base        140 k
 perl-DBI                    x86_64    1.627-4.el7              base        802 k
 perl-Data-Dumper            x86_64    2.145-3.el7              base         47 k
 perl-IO-Compress            noarch    2.061-2.el7              base        260 k
 perl-Net-Daemon             noarch    0.48-5.el7               base         51 k
 perl-PlRPC                  noarch    0.2020-14.el7            base         36 k
Updating for dependencies:
 mariadb-libs                x86_64    1:5.5.44-2.el7.centos    base        754 k

Transaction Summary
================================================================================
Install  1 Package  (+9 Dependent packages)
Upgrade             ( 1 Dependent package)

Total download size: 22 M
Is this ok [y/d/N]: y

...

Installed:
  mariadb-server.x86_64 1:5.5.44-2.el7.centos

Dependency Installed:
  mariadb.x86_64 1:5.5.44-2.el7.centos
  perl-Compress-Raw-Bzip2.x86_64 0:2.061-3.el7
  perl-Compress-Raw-Zlib.x86_64 1:2.061-4.el7
  perl-DBD-MySQL.x86_64 0:4.023-5.el7
  perl-DBI.x86_64 0:1.627-4.el7
  perl-Data-Dumper.x86_64 0:2.145-3.el7
  perl-IO-Compress.noarch 0:2.061-2.el7
  perl-Net-Daemon.noarch 0:0.48-5.el7
  perl-PlRPC.noarch 0:0.2020-14.el7

Dependency Updated:
  mariadb-libs.x86_64 1:5.5.44-2.el7.centos

Complete!
[root@server ~]#
```

MariaDB is now installed but (as it has been the case with nginx) neither is it running nor is it configured to start when our server (re)boots. To remedy this, we use the following commands:

```
[root@server ~]# systemctl enable mariadb.service
ln -s '/usr/lib/systemd/system/mariadb.service' '/etc/systemd/system/multi-
user.target.wants/mariadb.service'
[root@server ~]# systemctl start mariadb.service
[root@server ~]#
```

On a real production system, we would now run the command `/usr/bin/mysql_secure_installation` which allows us to set a database root password and remove the test database as well as anonymous users. Since we do not setup a production system in this book, you are free to run these steps but they are not required. If you do run this command, be sure to remember your database root password because we will need it later when we deploy our application.

3.4 Install a JavaScript runtime

In production mode, Rails needs a JavaScript runtime for the minification of our application's assets as part of their precompilation. We will install Node.js, which can be installed via the EPEL repository.

First we have to make the EPEL repository available to our server by installing the following package:

```
[root@server ~]# yum install epel-release

...

================================================================================
 Package                  Arch              Version           Repository      Size
================================================================================
Installing:
 epel-release             noarch            7-5               extras          14 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 14 k
Installed size: 24 k
Is this ok [y/d/N]: y

...

Installed:
  epel-release.noarch 0:7-5

Complete!
[root@server ~]#
```

When this is done, we can install Node.js like any other package:

```
[root@server ~]# yum install nodejs

...

================================================================================
 Package          Arch              Version                    Repository     Size
================================================================================
Installing:
 nodejs           x86_64            0.10.42-4.el7              epel          2.0 M
Installing for dependencies:
 libuv            x86_64            1:0.10.34-2.el7            epel           62 k
```

```
Transaction Summary
================================================================================
Install  1 Package (+1 Dependent package)

Total download size: 2.1 M
Installed size: 7.0 M
Is this ok [y/d/N]: y

...

Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7
Importing GPG key 0x352C64E5:
 Userid     : "Fedora EPEL (7) <epel@fedoraproject.org>"
 Fingerprint: 91e9 7d7c 4a5e 96f1 7f3e 888f 6a2f aea2 352c 64e5
 Package    : epel-release-7-5.noarch (@extras)
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7
Is this ok [y/N]: y

...

Installed:
  nodejs.x86_64 0:0.10.42-4.el7

Dependency Installed:
  libuv.x86_64 1:0.10.34-2.el7

Complete!
[root@server ~]#
```

3.5 Create an application user

We will now add a user to own all our Rails-related components. I usually call it `app` and I encourage you to do the same for the first time we set things up:

```
[root@server ~]# useradd app
[root@server ~]#
```

The next step is setting a password for this user – we will keep things simple my making username and password identical (even though the `passwd` command will complain for obvious reasons):

```
[root@server ~]# passwd app
Changing password for user app.
New password: app
BAD PASSWORD: The password is shorter than 8 characters
Retype new password: app
passwd: all authentication tokens updated successfully.
[root@server ~]#
```

3.6 Install Ruby

There are 3 ways to install Ruby on a server:

- Install from source
- Install with your distribution's package manager (yum in our case)
- Using a version manager

Each one has advantages and disadvantages.

Generally speaking, installing from source is the hardest way (especially for updating to a newer Ruby version) and has no real benefits. Installing through a package manager may seem comfortable but the provided Ruby versions are usually out of date and you cannot run multiple Ruby interpreters in parallel.

Using a version manager not only enables you to run several Ruby versions in parallel (to serve different Rails applications with different Ruby versions) but greatly simplifies the installation and update process.

When it comes to Ruby version management, you have two options: rbenv and RVM. Even though RVM is very popular I personally recommend rbenv. As Sam Stephenson (the author) explains on the project's Github page, rbenv is solely concerned with switching Ruby versions (whereas RVM does a lot more which complicates things).

To install rbenv and ruby-build (a project of the same author which simplifies compiling and installing Ruby versions that can be used as a rbenv plugin), basically all we need is git.

To enable ruby-build to compile a Ruby interpreter we have to make sure that we have some developer packages installed (the same ones that we would need for compiling Ruby from source ourselves) as well as some developer libraries.

First let's install all these required packages:

```
[root@server ~]# yum install gcc git libffi-devel libyaml-devel make openssl-
devel readline-devel zlib-devel
================================================================================
 Package                  Arch        Version                    Repository   Size
================================================================================
Installing:
 git                      x86_64      1.8.3.1-6.el7              updates     4.4 M
 libffi-devel             x86_64      3.0.13-16.el7              base         23 k
 libyaml-devel            x86_64      0.1.4-11.el7_0             base         82 k
 openssl-devel            x86_64      1:1.0.1e-51.el7_2.4        updates     1.2 M
 readline-devel           x86_64      6.2-9.el7                  base        138 k
 zlib-devel                x86_64     1.2.7-15.el7               base          50 k
Updating:
 gcc                       x86_64     4.8.5-4.el7                base          16 M
Installing for dependencies:
 keyutils-libs-devel       x86_64     1.5.8-3.el7                base          37 k
 krb5-devel                x86_64     1.13.2-10.el7              base         649 k
 libcom_err-devel          x86_64     1.42.9-7.el7               base          30 k
 libgnome-keyring          x86_64     3.8.0-3.el7                base         109 k
 libselinux-devel          x86_64     2.2.2-6.el7                base         174 k
 libsepol-devel            x86_64     2.1.9-3.el7                base          71 k
 libverto-devel            x86_64     0.2.5-4.el7                base          12 k
 libyaml                   x86_64     0.1.4-11.el7_0             base          55 k
 ncurses-devel             x86_64     5.9-13.20130511.el7        base         713 k
 pcre-devel                x86_64     8.32-15.el7                base         478 k
 perl-Error                noarch     1:0.17020-2.el7            base          32 k
 perl-Git                  noarch     1.8.3.1-6.el7              updates       53 k
 perl-TermReadKey          x86_64     2.30-20.el7                base          31 k
 rsync                     x86_64     3.0.9-17.el7               base         360 k
Updating for dependencies:
 cpp                       x86_64     4.8.5-4.el7                base         5.9 M
 krb5-libs                 x86_64     1.13.2-10.el7              base         843 k
 libffi                    x86_64     3.0.13-16.el7              base          30 k
 libgcc                    x86_64     4.8.5-4.el7                base          95 k
 libgomp                   x86_64     4.8.5-4.el7                base         130 k
 openssl                   x86_64     1:1.0.1e-51.el7_2.4        updates      711 k
 openssl-libs              x86_64     1:1.0.1e-51.el7_2.4        updates      951 k
 pcre                      x86_64     8.32-15.el7                base         418 k
 zlib                      x86_64     1.2.7-15.el7               base          89 k

Transaction Summary
================================================================================
Install  6 Packages (+14 Dependent packages)
Upgrade  1 Package  (+ 9 Dependent packages)

Total download size: 34 M
Is this ok [y/d/N]: y

...

Installed:
  git.x86_64 0:1.8.3.1-6.el7
  libffi-devel.x86_64 0:3.0.13-16.el7
  libyaml-devel.x86_64 0:0.1.4-11.el7_0
  openssl-devel.x86_64 1:1.0.1e-51.el7_2.4
  readline-devel.x86_64 0:6.2-9.el7
  zlib-devel.x86_64 0:1.2.7-15.el7

Dependency Installed:
  keyutils-libs-devel.x86_64 0:1.5.8-3.el7
  krb5-devel.x86_64 0:1.13.2-10.el7
  libcom_err-devel.x86_64 0:1.42.9-7.el7
  libgnome-keyring.x86_64 0:3.8.0-3.el7
  libselinux-devel.x86_64 0:2.2.2-6.el7
  libsepol-devel.x86_64 0:2.1.9-3.el7
  libverto-devel.x86_64 0:0.2.5-4.el7
  libyaml.x86_64 0:0.1.4-11.el7_0
  ncurses-devel.x86_64 0:5.9-13.20130511.el7
  pcre-devel.x86_64 0:8.32-15.el7
  perl-Error.noarch 1:0.17020-2.el7
  perl-Git.noarch 0:1.8.3.1-6.el7
  perl-TermReadKey.x86_64 0:2.30-20.el7
  rsync.x86_64 0:3.0.9-17.el7

Updated:
  gcc.x86_64 0:4.8.5-4.el7

Dependency Updated:
  cpp.x86_64 0:4.8.5-4.el7                    krb5-libs.x86_64 0:1.13.2-10.el7
  libffi.x86_64 0:3.0.13-16.el7               libgcc.x86_64 0:4.8.5-4.el7
  libgomp.x86_64 0:4.8.5-4.el7                openssl.x86_64 1:1.0.1e-51.el7_2.4
  openssl-libs.x86_64 1:1.0.1e-51.el7_2.4     pcre.x86_64 0:8.32-15.el7
  zlib.x86_64 0:1.2.7-15.el7

Complete!
[root@server ~]#
```

When this is done we can open a new login shell for our app user and install rbenv along with ruby-build:

```
[root@server ~]# su - app
[app@server ~]$ git clone  git://github.com/sstephenson/rbenv.git ~/.rbenv
Cloning into '/home/app/.rbenv'...
remote: Counting objects: 2495, done.
remote: Total 2495 (delta 0), reused 0 (delta 0), pack-reused 2495
Receiving objects: 100% (2495/2495), 449.29 KiB | 744.00 KiB/s, done.
Resolving deltas: 100% (1571/1571), done.
[app@server ~]$ git clone https://github.com/sstephenson/ruby-build.git
~/.rbenv/plugins/ruby-build
Cloning into '/home/app/.rbenv/plugins/ruby-build'...
remote: Counting objects: 5977, done.
remote: Total 5977 (delta 0), reused 0 (delta 0), pack-reused 5977
Receiving objects: 100% (5977/5977), 1.14 MiB | 917.00 KiB/s, done.
Resolving deltas: 100% (3401/3401), done.
[app@server ~]$ exit
logout
[root@server ~]#
```

Now we have to make sure that the rbenv environment is properly initialized when the app user logs in. This is usually done by placing the according lines (found in rbenv's installation instructions) in the app user's `~/.bash_profile` file but I like to handle this differently.

We change into the the directory `/etc/profile.d`, create a file called `rbenv.sh` and make it executable:

```
[root@server ~]# cd /etc/profile.d
[root@server profile.d]# touch rbenv.sh
[root@server profile.d]# chmod +x rbenv.sh
[root@server profile.d]#
```

All executable scripts withing this directory will be run when any user logs in so it is the ideal place to load the rbenv environment. Should you install rbenv for another user later, this user's rbenv environment will be loaded automatically.

In the script we just check whether a `.rbenv` directory exists in the current user's home directory and load rbenv if this is the case:

/etc/profile.d/rbenv.sh

```
#! /bin/sh

if [[ -e ${HOME}/.rbenv ]]; then
  export PATH="${HOME}/.rbenv/bin:${PATH}"
  eval "$(rbenv init -)"
fi
```

rbenv and ruby-build are now installed properly and will be laoded when the app user logs in the next time. To verfiy that, we open a login shell for the app user and run the `rbenv` command:

```
[root@server profile.d]# su - app
Last login: Wed Mar 16 11:44:06 CET 2016 on pts/0
[app@server ~]$ rbenv
rbenv 1.0.0-19-g29b4da7
Usage: rbenv <command> [<args>]

Some useful rbenv commands are:
   commands    List all available rbenv commands
   local       Set or show the local application-specific Ruby version
   global      Set or show the global Ruby version
   shell       Set or show the shell-specific Ruby version
   install     Install a Ruby version using ruby-build
   uninstall   Uninstall a specific Ruby version
   rehash      Rehash rbenv shims (run this after installing executables)
   version     Show the current Ruby version and its origin
   versions    List all Ruby versions available to rbenv
   which       Display the full path to an executable
   whence      List all Ruby versions that contain the given executable

See `rbenv help <command>' for information on a specific command.
For full documentation, see: https://github.com/rbenv/rbenv#readme
[app@server ~]$
```

When we run `rbenv versions` to list all available Ruby interpreters, we will se that there are no Rubies installed yet. Fortunately, installing a Ruby interpreter is ridiculously easy when using rbenv in combination with ruby-build.

While still logged in as app user, we wil install Ruby 2.3.0 (which is the latest version as I write this book – you can get a list of all available Rubies by running `rbenv install --list`). You will need a little patience because compiling Ruby from source (which is what ruby-build does in the background) takes some time:

```
[app@server ~]$ rbenv install 2.3.0
Downloading ruby-2.3.0.tar.bz2...
-> https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.0.tar.bz2
Installing ruby-2.3.0...
Installed ruby-2.3.0 to /home/app/.rbenv/versions/2.3.0

[app@server ~]$
```

We make this Ruby installation the app user's default by running the command `rbenv global 2.3.0`. This will make sure that the app user will always use Ruby 2.3.0 as long as no other Ruby version is specified on a per-project basis.

You can verify that everything worked as expected by running `rbenv versions` again and making sure that its output looks like this (the `*` indicates the global Ruby):

```
[app@server ~]$ rbenv global 2.3.0
* 2.3.0 (set by /home/app/.rbenv/version)
[app@server ~]$
```

3.6.1 Install bundler

A default ruby-build installation comes with a few gems:

```
[app@server ~]$ gem list --local

*** LOCAL GEMS ***

bigdecimal (1.2.8)
did_you_mean (1.0.0)
io-console (0.4.5)
json (1.8.3)
minitest (5.8.3)
net-telnet (0.1.1)
power_assert (0.2.6)
psych (2.0.17)
rake (10.4.2)
rdoc (4.2.1)
test-unit (3.1.5)
[app@server ~]$
```

Bundler is not among them so we will install it manually (you probably know that Rails depends on Bundler to manage its gem dependencies). We can skip the installation of a gem's documentation with the flags `--no-rdoc` and `--no-ri` which is what I like to do on staging and production machines:

```
[app@server ~]$ gem install bundler --no-rdoc --no-ri
Fetching: bundler-1.11.2.gem (100%)
Successfully installed bundler-1.11.2
1 gem installed
[app@server ~]$
```

We are done with the app user for now, so we exit from its shell:

```
[app@server ~]$ exit
logout
[root@server profile.d]# cd
[root@server ~]
```

3.7 Take a snapshot

We are through with this chapter so do not forget to take a snapshot:

```
[root@server ~]# exit
logout
Connection to byors.local closed.
[michael@development byors]$ vagrant snapshot save "End of chapter 3"
==> default: Snapshotting the machine as 'End of chapter 3'...
==> default: Snapshot saved! You can restore the snapshot at any time by
==> default: using `vagrant snapshot restore`. You can delete it using
==> default: `vagrant snapshot delete`.
[michael@development byors]$
```

4 Cold deploy

In this chapter we will cold deploy a sample application to our server which means getting it to run for the first time (think of Capistrano's `deploy:cold` task if you have ever used it). There are quite some steps involved in doing so, but it is a fairly logical step-by-step process.

The application we will deploy is the simplest one you can imagine – just one model and one controller, managed by a scaffold. It has actually been built for my book Efficient Rails DevOps (hence the name of it's repository) but it will serve the purpose of this book equally well.

4.1 Directory structure

We will first prepare the directory structure on our server by logging in as root and create the base directory for all our applications:

```
[michael@development byors]$ ssh root@byors.local
root@byors.local's password: vagrant
Last login: Wed Mar 16 11:54:22 2016
[root@server ~]# mkdir -p /var/www
[root@server ~]#
```

We will use a repository-based deployment scenario, where you defne one branch in your repository which holds only commits of production-ready code (I use the `master` branch for that). Our application server's workers get their code directly from the repository and since they keep a copy of the whole application in memory, it does not matter how long it takes to pull from the remote. The new updated code is not loaded until we tell the application server to do so.

Besides (even though we will not cover that topic in this book), a repository-based deployment scenario greatly simplifes zero-downtime-deploys – one of the things where the Unicorn application server really shines.

What we need is a directory to hold our application's codebase (we'll simply call it `application`) and a `shared` directory containing subdirectories for configuration, logs, pids and temporary files.

We will end up with a directory structure that looks like this:

- myapp
  - application
  - shared
    - config
    - log
      - archive
    - pids
    - tmp
      - cache

We create this directory structure with the following commands (note that we are making the app user the owner of `/var/www/myapp` and all directories beyond):

```
[root@server ~]# cd /var/www
[root@server www]# mkdir myapp
[root@server www]# cd myapp
```

```
[root@server myapp]# mkdir application shared
[root@server myapp]# cd shared
[root@server shared]# mkdir -p config log/archive pids tmp/cache
[root@server shared]# cd ../..
[root@server www]# chown -R app myapp
[root@server www]#
```

4.2 Clone the app's codebase

We clone our application's repository into the `/var/www/myapp/application` directory. Before doing so, we switch to our app user which spares us from manually adjusting the owner of all cloned files and directories:

```
[root@server www]# su - app
Last login: Wed Mar 16 15:00:00 CET 2016 on pts/0
[app@server ~]$ cd /var/www/myapp/application
[app@server application]$ git clone https://github.com/relativkreativ/erdo-
sample-app.git .
Cloning into '.'...
remote: Counting objects: 109, done.
remote: Total 109 (delta 0), reused 0 (delta 0), pack-reused 109
Receiving objects: 100% (109/109), 23.00 KiB | 0 bytes/s, done.
Resolving deltas: 100% (14/14), done.
[app@server application]$
```

4.3 Set environment variables

We want to run our application in production mode on our virtual machine. To make sure that none of our applications is started in development mode by accident, we need a way to set the `RAILS_ENV` environment variable system-wide.

In addition, we need an environment variable called `SECRET_KEY_BASE` since starting with version 4 Rails no longer stores the production key base directly in the file `config/secrets.yml` but reads it from an environment variable. You can generate a key on your development machine using the rake task `rake secret` (or just use mine to try things out).

A good place to set these two variables is the `/etc/profile.d` directory (where we also initialized rbenv). We go there as root, create a file called `rails.sh` and make it executable:

```
[app@server ~]$ exit
logout
[root@server www]# cd /etc/profile.d
[root@server profile.d]# touch rails.sh
[root@server profile.d]# chmod +x rails.sh
[root@server profile.d]#
```

Then we edit the file to look like this (paste in your own secret key base if you generated one and be careful to not introduce any new linebreaks – the key has to go on one line):

/etc/profile.d/rails.sh

```
export RAILS_ENV=production
export
SECRET_KEY_BASE=826afb1bf8b20aa79e1b1e96790f9de0bc003809cedb491db74fd8eee5143
69c9380b0a64f965f9c925676fbe44b269ba6d05bf7224796c7e7834689364d1510
```

4.4 Configure our application server

As with every component in our Rails stack, we have different application servers to choose from.

The most popular ones are Passenger and Unicorn. Passenger surely has its advantages but we will use Unicorn – even if not covered in this book, it enables us zero-downtime deploys (which Passenger does only with its paid version), has a more predictable memory footprint and is more flexible in our environment.

We do not have to install it explicitly but via our application's bundle. So if you are deploying your own application, make sure to include `gem 'unicorn'` in its Gemfile for the production environment.

Once again we switch to our app user and create the file holding the Unicorn configuration for our application:

```
[root@server profile.d]# su - app
Last login: Wed Mar 23 10:20:20 CET 2016 on pts/0
[app@server ~]$ cd /var/www/myapp/shared/config
[app@server config]$ touch unicorn.rb
[app@server config]$
```

This is how our configuration looks:

/var/www/myapp/shared/config/unicorn.rb

```
root = '/var/www/myapp'

worker_processes 3
user 'app'
preload_app true
timeout 30

listen '192.168.0.100:8080'
pid "#{root}/shared/pids/unicorn.pid"
working_directory "#{root}/application"
stdout_path "#{root}/shared/log/unicorn.log"
stderr_path "#{root}/shared/log/unicorn_error.log"
```

The configuration options should be pretty self-explanatory.

What's interesting is the `listen` directive: It tells Unicorn's master process to wait for incoming connections on port 8080 of the IP address our server uses. We will later configure our webserver to forward requests to this port.

4.5 Link shared files and directories

The files and directories in our `shared` directory exist for one reason: They must survive deploys of our application (this includes technical stuff like pids and temporary files but it is also the right place for user-generated data like uploaded images).

Since the `shared` directory is out of reach for our application, we have to link its subdirectories to the appropriate places in our application's directory structure. In order to do that, we first have to remove the original files which will be replaced.

We do this as app user:

```
[app@server config]$ cd /var/www/myapp/application
[app@server application]$ rm -rf log
[app@server application]$ ln -s /var/www/myapp/shared/log ./log
[app@server application]$ rm -rf tmp
[app@server application]$ ln -s /var/www/myapp/shared/tmp ./tmp
[app@server application]$ rm -f config/database.yml
[app@server application]$ ln -s /var/www/myapp/shared/config/database.yml
./config/database.yml
[app@server application]$ ln -s /var/www/myapp/shared/config/unicorn.rb
./config/unicorn.rb
[app@server application]$
```

4.6 Configure SELinux

SELinux (Security-Enhanced Linux) is a kernel security module that provides a mechanism for supporting access control security policies, including mandatory access controls – think of it as kind of a firewall on application level.

No matter what you have heard about SELinux around the web – you absolutely should be using it.

When used correctly, it greatly reduces your server's surface of vulnerability – but it can also lead to hours of troubleshooting when you do not know how it works. That's why far too many people advise you to simply turn it off.

SElinux is enabled on CentOS systems per default but some hosting providers provide base systems with SELinux disabled. You should consider enabling it.

It is enabled on the Vagrant box coming with this book and we need to make two adjustments to how SELinux works in order to make it play nicely with our application.

The first thing we will do is setting a different SELinux type for our application's log files. SELinux types are set depending on the location of files and directories and since our application's logs are not kept in our system's default log directory (`/var/log`) they have the wrong SELinux type (`httpd_sys_content_t` instead of `var_log_t` – you can view the SELinux context of files by using the `ls` command's `-Z` flag) making it impossible to rotate them with logrotate (should you wish to do that).

In order to manipulate SELinux rules we need to have the `policycoreutils-python` package installed so will will install it now:

```
[app@server application]$ exit
logout
[root@server profile.d]# yum install policycoreutils-python
================================================================================
 Package                      Arch         Version              Repository  Size
================================================================================
Installing:
 policycoreutils-python       x86_64       2.2.5-20.el7         base       435 k
Installing for dependencies:
 audit-libs-python            x86_64       2.4.1-5.el7          base        69 k
 checkpolicy                  x86_64       2.1.12-6.el7         base       247 k
 libcgroup                    x86_64       0.41-8.el7           base        64 k
 libsemanage-python           x86_64       2.1.10-18.el7        base        94 k
 python-IPy                   noarch       0.75-6.el7           base        32 k
 setools-libs                 x86_64       3.3.7-46.el7         base       485 k
```

```
Updating for dependencies:
 libsemanage                   x86_64        2.1.10-18.el7        base        123 k
 policycoreutils               x86_64        2.2.5-20.el7         base        803 k

Transaction Summary
================================================================================
Install  1 Package  (+6 Dependent packages)
Upgrade             ( 2 Dependent packages)

Total download size: 2.3 M
Is this ok [y/d/N]: y

...

Installed:
  policycoreutils-python.x86_64 0:2.2.5-20.el7

Dependency Installed:
  audit-libs-python.x86_64 0:2.4.1-5.el7
  checkpolicy.x86_64 0:2.1.12-6.el7
  libcgroup.x86_64 0:0.41-8.el7
  libsemanage-python.x86_64 0:2.1.10-18.el7
  python-IPy.noarch 0:0.75-6.el7
  setools-libs.x86_64 0:3.3.7-46.el7

Dependency Updated:
  libsemanage.x86_64 0:2.1.10-18.el7    policycoreutils.x86_64 0:2.2.5-20.el7

Complete!
[root@server ~]#
```

When this is done, we can create a rule which applies the SELinux type `var_log_t` to all files and directories under our application's `shared/log` directory and relabel them based on this new rule:

```
[root@server profile.d]# semanage fcontext -a -t var_log_t
"/var/www/myapp/shared/log(/.*)?"
[root@server profile.d]# restorecon -R /var/www/myapp
[root@server profile.d]#
```

The second thing we need to do is activating a SELinux boolean to allow our nginx webserver to later connect to the port our Unicorn application server is listening. A SELinux boolean includes a set of policy rules to allow certain activities. The one we need to activate is called `httpd_can_network_connect`:

```
[root@server profile.d]# setsebool -P httpd_can_network_connect=1
[root@server profile.d]#
```

4.7 Install our application's bundle

If we were simply trying to install our application's bundle now, the native extensions for the mysql2 gem would not be built because we do not have the necessary development package installed.

So we first do that:

```
[root@server profile.d]# yum install mariadb-devel

...

================================================================================
 Package                Arch          Version                      Repository  Size
================================================================================
Installing:
 mariadb-devel          x86_64        1:5.5.44-2.el7.centos        base       748 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 748 k
Installed size: 3.3 M
Is this ok [y/d/N]: y

...

Installed:
  mariadb-devel.x86_64 1:5.5.44-2.el7.centos

Complete!
[root@server profile.d]#
```

Furthermore, Bundler needs the patch package in order to work, so we install that too:

```
[root@server profile.d]# yum install patch
================================================================================
 Package            Arch              Version                   Repository      Size
================================================================================
Installing:
 patch              x86_64            2.7.1-8.el7               base           110 k

Transaction Summary
================================================================================
Install  1 Package
```

```
Total download size: 110 k
Installed size: 210 k
Is this ok [y/d/N]: y

...

Installed:
  patch.x86_64 0:2.7.1-8.el7

Complete!
[root@server profile.d]#
```

When this is done we can install our application's bundle. We have to do this as app user because our app will later be run as this user (and it is the only one on our system with a working Ruby environment):

```
[root@server profile.d]# su - app
Last login: Thu Mar 17 14:25:39 CET 2016 on pts/1
[app@server ~]$ cd /var/www/myapp/application
[app@server application]$ bundle install --without=development test

...

Bundle complete! 14 Gemfile dependencies, 51 gems now installed.
Gems in the groups development and test were not installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
[app@server application]$
```

4.8 Prepare the database

We have to connect to our database as root since it's the only database user at the moment (don't confuse MariaDB's root user with the root user of our server). If you have not set a database root password when installing MariaDB, simply press `Return` when prompted for the password (otherwise enter it):

```
[app@server application]$ mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 2
Server version: 5.5.44-MariaDB MariaDB Server

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

First we create a database for our application and name it like the application itself:

```
MariaDB [(none)]> CREATE DATABASE myapp;
Query OK, 1 row affected (0.00 sec)

MariaDB [(none)]>
```

Then we create a database user (which will also be named like our application) and restrict access to just this one database (again, `myapp` is an extremely bad password so you should really generate a strong one if you do this on a production machine). When we're done, we can exit the database shell:

```
MariaDB [(none)]> CREATE USER 'myapp'@'localhost' IDENTIFIED BY 'myapp';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> GRANT ALL PRIVILEGES ON myapp.* TO 'myapp'@'localhost';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> EXIT
Bye
[app@server application]$
```

Our database is ready now.

Loading our application's database schema is easy because there is a dedicated rake task for it. Note that we prepend every call to an executable installed with a gem with `bundle exec` – this ensures that Bundler picks the right version of the executable (deploying another Rails application later may add another version of an already existing executable).

If you want to know more about how Bundler manages different gem versions, I wrote an article about that topic some time ago.

A lot of developers load their database schema by running the `db:migrate` rake task (I did so myself in the previous version of this book). This can lead to serious problems and it's way better to use the dedicated rake task `db:schema:load`. As a rule of thumb, use `db:migrate` for deploying and `db:schema:load` for cold deploying.

Before we can load our database schema, we have to tell our application about the database. We have already created a `database.yml` file and symlinked it into our `application` directory (thus ignoring the `database.yml` file that comes with our application, should there be one), but it is still empty.

Now we edit it to include our application's connection settings:

/var/www/myapp/shared/config/database.yml

```
production:
  adapter: mysql2
  encoding: utf8
  reconnect: true
  database: myapp
  username: myapp
  password: myapp
```

Once we have saved this file, we can load our database schema into the database:

```
[app@server application]$ bundle exec rake db:schema:load
-- create_table("customers", {:force=>:cascade})
   -> 0.0109s
-- initialize_schema_migrations_table()
   -> 0.0251s
[app@server application]$
```

4.9 Precompile assets

We have to precompile our application's assets to enable our webserver to serve them as static assets. This is easy as there is a dedicated rake task for it:

```
[app@server application]$ bundle exec rake assets:precompile
I, [2016-03-18T00:51:43.388927 #8024]  INFO -- : Writing
/var/www/myapp/application/public/assets/application-
b598aa7e82e5647cd54ae6f409306995766dd0ce1af586bab9ebe84804d0eec0.js
I, [2016-03-18T00:51:43.407557 #8024]  INFO -- : Writing
/var/www/myapp/application/public/assets/application-
0723cb9a2dd5a514d954f70e0fe0b89f6f9f1ae3a375c182f43b5f2b57e9c869.css
[app@server application]$
```

4.10 Set SELinux types and contexts

We introduced quite a few new files to our system. To make sure that all of them have the proper SELinux types and contexts, we run the following command at the end of our cold deploy process:

```
[app@server application]$ restorecon -R /var/www/myapp
[app@server application]$
```

4.11 Start the application server

To start the application server, we have to tell it in which mode and point it to a configuration file (the `unicorn.rb` file we created earlier). The `-D` flag makes it run as a daemon:

```
[app@server application]$ bundle exec unicorn -D -c
/var/www/myapp/shared/config/unicorn.rb -E production
[app@server application]$
```

When we are listing all Unicorn processes our system is running, we can see that the Unicorn master process has spawned three worker processes:

```
[app@server application]$ ps aux | grep unicorn
app       2772  0.8  3.9 396792 73624 ?        Sl   11:04   0:00 unicorn master
-D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       2776  0.0  3.8 396792 70904 ?        Sl   11:04   0:00 unicorn
worker[0] -D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       2779  0.0  3.8 396792 70900 ?        Sl   11:04   0:00 unicorn
worker[1] -D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       2782  0.0  3.8 396792 70900 ?        Sl   11:04   0:00 unicorn
worker[2] -D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       2789  0.0  0.0 112640   960 pts/0    R+   11:05   0:00 grep
--color=auto unicorn
[app@server application]$
```

4.12 Configure a virtual host for nginx

At this point, our nginx webserver knows nothing about our application and that it should be served when someone visits http://byors.local or http://www.byors.local. It will always serve its default welcome page instead.

To change that, we have to create a virtual host for our application as the last step for our cold deploy routine.

nginx automatically includes all configuration files ending with `.conf` in the `/etc/nginx/conf.d` directory into its configuration. So we go there as root user and create a file called `myapp.conf`:

```
[app@server application]$ exit
logout
[root@server profile.d]# cd /etc/nginx/conf.d
[root@server conf.d]# touch myapp.conf
```

Then we edit this file to look like this:

/etc/nginx/conf.d/myapp.conf

```
upstream myapp {
  server 192.168.0.100:8080;
}

server {
  listen 80;
  server_name www.byors.local;
  return 301 http://byors.local$request_uri;
}

server {
  listen 80;
  server_name byors.local;
  root /var/www/myapp/application/public;
  client_max_body_size 4G;
  keepalive_timeout 5;
  access_log /var/log/nginx/access.myapp.log;

  location ~ ^/assets/ {
    root /var/www/myapp/application/public;
    gzip_static on;
    expires max;
    add_header Cache-Control public;
    add_header ETag "";
    break;
26    }
27
28    location / {
29      try_files $uri @myapp;
30    }
31
32    location @myapp {
33      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
34      proxy_set_header Host $http_host;
35      proxy_redirect off;
36      proxy_pass http://myapp;
37    }
38  }
```

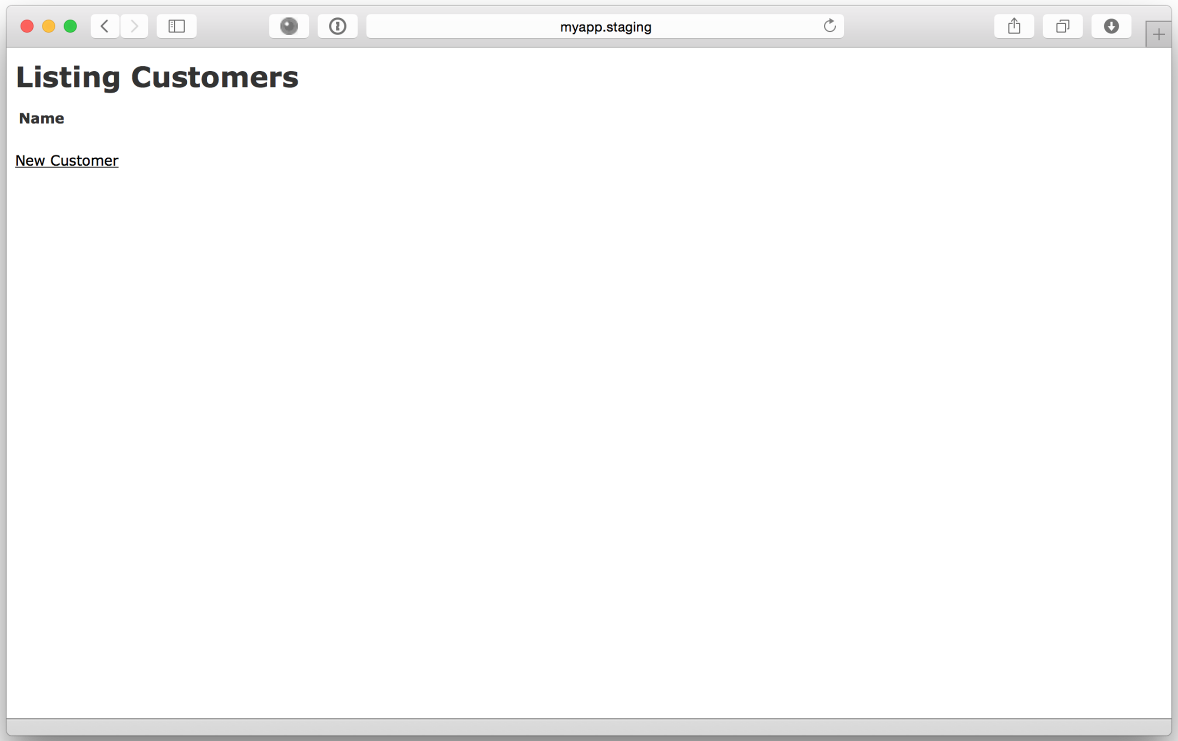
We have defined an upstream resource for our Unicorn application server where nginx can redirect Rails requests to (line 1 to 4), a vhost to redirect www-requests to their non-www counterpart (line 5 to 9) and the main configuration (line 11 to 38).

The main configuration sets the document root and logfile path, handles requests to our application's `assets` directory (by serving them directly, compressed and with a far-future expiration date). Other requests are then forwarded to our Unicorn application server.

We need to restart nginx for the changes to take effect:

```
[root@server conf.d]# systemctl restart nginx.service
[root@server conf.d]#
```

If you now browse to http://byors.local or http://www.byors.local now, you will be greeted by the Rails application we have just cold deployed:



Listing Customers

Name

New Customer

Congratulations – we're up and running!

4.13 Take a snapshot

As always, take a snapshot of our virtual machine:

```
[root@server conf.d]# exit
logout
Connection to byors.local closed.
[michael@development byors]$ vagrant snapshot save "End of chapter 4"
==> default: Snapshotting the machine as 'End of chapter 4'...
==> default: Snapshot saved! You can restore the snapshot at any time by
==> default: using `vagrant snapshot restore`. You can delete it using
==> default: `vagrant snapshot delete`.
[michael@development byors]$
```

5 Deploy

5.1 Install a Unicorn service

There is one thing that makes deploying a new version of our application tedious right now: Starting, reloading and stopping of our application server.

In the previous chapter we already started Unicorn manually – now let's see how to stop it:

We have to lookup our Unicorn master's PID (which is stored in `/var/www/myapp/shared/pids/unicorn.pid`) and send this process the `QUIT` signal. Tedious but it works.

Let's do this once to start with a clean environment.

```
[michael@development byors]$ ssh root@byors.local
root@byors.local's password: vagrant
Last login: Wed Mar 23 10:16:35 2016 from 192.168.0.118
[root@server ~]# kill -s QUIT $(cat /var/www/myapp/shared/pids/unicorn.pid)
[root@server ~]#
```

We already used systemd services to enable and start nginx and MariaDB. Unfortunately, Unicorn does not come with a predefined systemd service so we will build one ourselves.

While still being logged in as root, we create the following file:

```
[root@server ~]# cd /usr/lib/systemd/system
[root@server system]# touch unicorn-myapp.service
[root@server system]# chmod 0755 unicorn-myapp.service
[root@server system]#
```

systemd services are pretty standardized – we just have to fill in the commands needed to start, reload and stop the application we are managing:

/usr/lib/systemd/system/unicorn-myapp.service

```
[Unit]
Description=myapp application server
After=network.target

[Service]
Type=forking
PIDFile=/var/www/myapp/shared/pids/unicorn.pid
ExecStart=/usr/bin/bash -lc "cd /var/www/myapp/application &&
/home/app/.rbenv/shims/bundle exec unicorn -D -c
/var/www/myapp/shared/config/unicorn.rb -E production"
```

```
ExecReload=/bin/kill -s USR2 $MAINPID
ExecStop=/bin/kill -s QUIT $MAINPID

[Install]
WantedBy=multi-user.target
```

The most interesting part of this configuration file happens on line 8:

We start Unicorn from a new login shell (`/usr/bin/bash -l`), change into our application's directory and hijack the app user's `rbenv` directory to use the proper version of the `unicorn` executable for our application (rbenv's `shims` directory holds symlinks to all our gems' executables matching our current Ruby version).

This is necessary because this systemd service is managed via the `sysctl` command as root.

After saving this file, we are able to manage our application's Unicorn the same way we did with nginx and MySQL. So while we're at it, we first make our application server start when our server (re)boots:

```
[root@server system]# systemctl enable unicorn-myapp.service
ln -s '/usr/lib/systemd/system/unicorn-myapp.service'
'/etc/systemd/system/multi-user.target.wants/unicorn-myapp.service'
[root@server system]#
```

We want to execute our deploys as app user, so we need to enable the app user to manage our service by using `sudo` to run the appropriate commands with root privileges. To configure this, we need to edit our sudoers configuration.

The main sudoers configuration file (`/etc/sudoers.conf`) includes all files in the `/etc/sudoers.d` directory so we keep things neatly organized by adding a file in this directory:

```
[root@server system]# cd /etc/sudoers.d
[root@server sudoers.d]# touch myapp
[root@server sudoers.d]#
```

This file has the following contents:

/etc/sudoers.d/myapp

```
app ALL= NOPASSWD: /usr/bin/systemctl * unicorn-myapp.service

Defaults: app env_reset
Defaults: app env_keep += "RAILS_ENV SECRET_KEY_BASE"

```

This configuration makes sure that the app user is able to run the commands needed to start, reload and stop our application server using `sudo` without providing a password

Furthermore, it ensures that the app user's `RAILS_ENV` and `SECRET_KEY_BASE` environment variable are preserved when temporarily becoming root (they are not by default).

We can try out things now just for fun. We start a login shell as app and start our application server with simple command:

```
[root@server sudoers.d]# su - app
Last login: Wed Mar 23 11:43:40 CET 2016 on pts/0
[app@server ~]$ sudo systemctl start unicorn-myapp.service
[app@server ~]$
```

Our Unicorn workers are started:

```
[app@server ~]$ ps aux | grep unicorn
root      3629 14.4  3.9 397276 72140 ?        Sl   11:51   0:00 unicorn master
-D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       3633  0.2  3.7 396720 68952 ?        Sl   11:51   0:00 unicorn
worker[0] -D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       3635  0.2  3.7 396720 68952 ?        Sl   11:51   0:00 unicorn
worker[1] -D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       3638  0.2  3.7 396720 68956 ?        Sl   11:51   0:00 unicorn
worker[2] -D -c /var/www/myapp/shared/config/unicorn.rb -E production
app       3643  0.0  0.0 112640   960 pts/0    R+   11:51   0:00 grep
--color=auto unicorn
[app@server ~]$
```

Stopping them is equally easy:

```
[app@server ~]$ sudo systemctl stop unicorn-myapp.service
[app@server ~]$ ps aux | grep unicorn
app       3650  0.0  0.0 112640   960 pts/0    R+   11:52   0:00 grep
--color=auto unicorn
[app@server ~]$
```

With this systemd service in place, deploying a new release of our application gets a lot easier as we will see in the next section.

5.2 The steps to a new release

Pushing a new release of our application requires the following steps:

- Reset the server's repository and pull the latest changes from its origin
- Shut down the application server
- Symlink our shared files and directories
- Update our application's bundle
- Precompile assets
- Migrate the database
- Start the application server

We will now execute these steps, one after another.

First, we update our applications codebase by pulling the latest changes from our repository's remote. Before we do that, we have to discard all local changes (which means deleting all symlinks we created and restore the original files and directories). Do not worry about downtime, our application server's worker each hold a copy of our codebase in memory, so our application will still be served as long as we do not touch our Unicorn master process.

```
[app@server ~]$ cd /var/www/myapp/application
[app@server application]$ git reset --hard
...
[app@server application $] git pull
...
[app@server application]$
```

Then we reinstall the symlinks to our shared files and directories:

```
[app@server application]$ rm -rf log
[app@server application]$ ln -s /var/www/myapp/shared/log ./log
[app@server application]$ rm -rf tmp
[app@server application]$ ln -s /var/www/myapp/shared/tmp ./tmp
[app@server application]$ rm -f config/database.yml
[app@server application]$ ln -s /var/www/myapp/shared/config/database.yml
./config/database.yml
[app@server application]$ rm -f config/unicorn.rb
[app@server application]$ ln -s /var/www/myapp/shared/config/unicorn.rb
./config/unicorn.rb
```

```
[app@server application]$
```

Then we install our application's bundle. This will install any new gem dependencies.

```
[app@server application]$ bundle install --without=development test
...
[app@server application]$
```

We precompile our application's assets and remove old ones which are no longer in use:

```
[app@server application]$ bundle exec rake assets:precompile
...
[app@server application]$ bundle exec rake assets:clean
...
[app@server application]$
```

Apply the proper SELinux contexts to all new files:

```
[app@server application]$ restorecon -R /var/www/myapp
[app@server application]$
```

In the next step we will migrate our application's database. To protect our database's integrity (and to make sure that our application does not run with a database schema that's too new because our application still runs on old code) we will stop our application server before the migration:

```
[app@server application]$ sudo systemctl stop unicorn-myapp.service
[app@server application]$
```

Now we can migrate:

```
[app@server application]$ bundle exec rake db:migrate
...
[app@server application]$
```

When this last step is completed, we can start our application server which will now pickup the changes to our codebase:

```
[app@server application]$ sudo systemctl start unicorn-myapp.service
[app@server application]$
```

Admittedly, there is no new code in our application's repository so carrying out these steps actually does nothing more than stopping and restarting our application server.

However, I encourage you to try out the things you have learned in this book with your own applications!

6 Epilogue

Congratulations, we're through!

As time is our most valuable asset, I want to sincerely thank you for taking the time to read and work through this book as well as for trusting me with your email address.

What you have just finished is the completely rewritten second edition of this book so it may still be a little rough around the edges. I am obsessed with the quality of my content so if you found any issues (or just want to tell me what you think about the book) please shoot a message to michael.trojanek@relativkreativ.at anytime!

Of course we hardly scratched the surface of the complex topic of running Rails applications in the public. While hopefully this book gave you some valuable insights on how to provision a Rails server and deploy an application, you would not want to apply the steps outlined in this book to your production environment as is.

Before doing so, you should have built a proven, versioned and automated process to provision your servers and deploy your applications.

If this is the way you want to go, you may be interested in Efficient Rails DevOps, the second book that I wrote. Compared to this book, it takes care of everything from top to bottom (including, but not limited to serving your applications encrypted via SSL, making use of lightning fast zero-downtime deploys and preparing your server to run more than one Rails application, even with different Ruby versions) leveraging the power of Ansible (a fantastic configuration management tool) to build a completely automated system to manage your Rails environment.

Thanks a lot for your support and now back to building (and releasing) awesome Rails-backed products!