Table Of Contents

Introduction

Thank you for reading Advanced Web Applications! You are reading this book not only because you want to become more familiar with the PHP framework Symfony, but also because you want to gain a list of skills you can directly apply to a wide variety of web development tasks. Not only will this book show you how to use Symfony at an expert level, but you will learn how to configure a production server, develop Symfony applications on a team, secure your server with SSL, SSH, and `iptables`, deploy your code without downtimes, and many more advanced topics.

About Me

My name is Vic Cherubini and I have been programming for the web since 1999. I was fortunate to grow up in an environment that fostered a passion for computers: my father owns a software company in Houston, TX. I was thirteen years old when the web really started to become popular in 1997 and I spent countless hours at my father's office bugging the other developers to teach me HTML.

In 1999 I finally picked up some Perl, and quickly moved into learning PHP. Understanding PHP opened up a whole new world of opportunities for me. Finally, I had an easy to understand programming language that could connect to a database! It was quite earth shattering.

The last fourteen years have taught me a lot. I consider myself an expert level developer, and my goal with this book is to make you a better programmer. As for my official credentials, I have a Bachelors of Science in Computer Science from the University of Texas at Dallas. I started programming PHP in 1999 and have had a job doing that in one form or another since then.

I currently live in Dallas, TX and run a small software consulting company, *Bright March*, fulltime. We have built a variety of web applications, almost entirely in Symfony. With each application, my knowledge of Symfony has become stronger, and I am finally at a point where I feel comfortable writing this book. I have helped author other technical books in the past; this is the first commercial book I have authored entirely — with plenty of help, naturally.

It is my sincere hope that you become not only an expert level Symfony developer by the conclusion of this book, but also that you become a well rounded engineer that has the ability to handle any problem thrown their way.

About You

This book is aimed at the Symfony or PHP developer that wants to become a better, more well-rounded programmer. Many pieces of this book will not be only about Symfony. Thus, you should have a solid grasp on PHP already. You should be relatively comfortable from the command line, and I highly recommend you have a Unix based computer to work on (however, we will be using Vagrant to create a normalized development environment).

Additionally, you should have a basic understanding of how Model-View Controller (MVC) frameworks work (don't worry as much about Object Relational Mappers (ORM) — we'll discuss Doctrine in depth). Ideally you are aware of how Symfony is set up; even better if you have already installed and started to tinker with it. While you can certainly gain a lot from this book without having a Symfony project under your belt, there may be some areas where the Symfony documentation explains a basic subject in more detail (for example, understanding what a Bundle is, or their default directory structure).

Sample Symfony Applications

Symfony2 — Symfony throughout the rest of the book unless noted otherwise — was released on July 28, 2011. The company I worked for at the time had several Symfony 1.0 and 1.4 applications in production (and was committed to using Symfony in the future), so I decided to give Symfony a shot for a side project. Before using Symfony, I preferred to write my own frameworks. Like many PHP developers, I always felt using a mainstream framework meant my application was going to be slow. To date I have written three separate MVC frameworks (Artisan System, Jolt), and Kin, and one ORM (DataModeler).

Writing my own MVC frameworks certainly taught me a lot on how frameworks are structured, so I do not consider the time I put in wasted effort, but in the end, it is always a much better idea to use a popular framework *when the application suites it*.

While I was certainly productive using my own frameworks, they would always be missing something. Even though I understood everything under the hood, development ended up taking longer because I would have to implement some piece of boilerplate code.

Symfony was a breath of fresh air for me. It handled a tremendous amount of boilerplate code very well, and made development happen at a rapid pace. Symfony is also useful at multiple levels of PHP expertise. Junior developers can use it knowing it takes care of a lot of complex tasks, and more senior developers can take advantage of all of the different configuration options it offers.

By introducing the Bundle system, Symfony finally made library and code reuse easy and intuitive. Combine the Bundle system with Composer and Packagist, and you have a very robust environment for building complex applications without duplicating code.

The first Symfony application I built was quite complex. It handled moving thousands of e-commerce order data a day to different warehouses using a REST API. That initial project used Resque, Redis, PostgreSQL, a Ruby program named God, and of course PHP. I was able to thoroughly test it with PHPUnit and it performed very, very well. I was really proud of my work and impressed with what Symfony could do.

From there, I went on to build several other REST APIs using an open source bundle I wrote. Eventually, I started converting all of Bright March's projects from my custom frameworks to Symfony with great success. It was during these transitions that I was able to hone how to configure, build, and deploy a Symfony application. The majority of this book will be taught using that knowledge.

When Not To Use Symfony

As I mentioned previously, Symfony is an amazing framework when the application suites it. When does using Symfony (or any framework for that matter) not make sense? There are a few situations.

Small Application

Symfony is a large, complex framework. If your application is very small, you do not need database access, or it will not be in production a long time, Symfony is probably not a good choice. A small PHP script or static HTML can serve your needs just fine.

Absolute Performance

If you need a dynamic application where performance is paramount above all else, then you might look into writing it in "raw" PHP — or ditching PHP altogether and using a faster language like Java, C++, or Go.

In a previous job, we had a very small API that delivered a small XML payload to millions of Android phones running our software alerting them if an update was available. Each request used exactly two database queries, and each payload was around 400 bytes in size. We served around 540,000,000 requests a month using a very small API written using "raw" PHP.

No Control Over Production Environment

Using Symfony on a managed server or hosting environment that you do not control entirely is not something I would advise. This means your "unlimited" host you pay almost no money for is not suited for running a Symfony application. You need root access to the production machine. However, Symfony works great in a virtualized environment. In fact, all Bright March's applications are hosted using virtual machines. And of course, Symfony works wonderful if you own or control the hardware it runs on too.

MajorAuth

The application you will build throughout this book is named MajorAuth. Within Bright March, we have a whole series of projects that start with the prefix "Major" and this is one of them.

MajorAuth is a user management as a service REST API. MajorAuth is similar (though less feature-filled) to the services Mozilla Persona, Stormpath, and Authy (among many others).

Imagine you manage ten e-commerce sites and each of them has their own authentication and user management code. You discover a bug in that code which means that you have to go around to each implementation and fix the bug. MajorAuth fixes that by storing all account data in one central location. Each site makes HTTP requests to the API to manage users and perform authentication. If a bug with authentication were to pop-up, you would only have to fix it in one place rather than all ten.

This book will take you through building a MajorAuth clone from scratch. Each chapter will build upon the previous so by the end of the book you have a complete application you can put into production. It is highly recommended you read each chapter sequentially.

MajorAuth comes with two main components: the REST API and admin panel. The REST API is an HTTP API that primarily uses JSON as the content type, and the admin panel is a basic HTML frontend that allows you to manage account data visually.

Even if the MajorAuth product does not solve an immediate problem of yours, the skills you will learn during its development throughout this book will make you a better programmer.

Large Code Samples

We will be going through a lot of code in this book. Large code samples do not lend themselves well to traditional print layouts. To remedy this, a link to a GitHub Gist will be used. This gives us the added benefit that readers of the book can comment on the code. Additionally, its history will be automatically preserved by GitHub so you can watch as it progresses with multiple versions of the book.

Code samples to GitHub Gists are hyperlinks in the PDF and HTML versions of this book and will resemble the style below.

```
https://gist.github.com/leftnode/6367991
```

Lets get started!

Configuring Your Environment

It is time to get started! The very first thing we must do is to ensure you have a capable environment to develop in.

To do this, we are going to use a piece of software named Vagrant to normalize your environment and my environment. This way, we guarantee they are identical and issues of "it works on my machine!" will be minimized.

Vagrant is a simple yet incredibly useful piece of software. Essentially, it is a command line wrapper to a virtual machine hypervisor (VirtualBox by default) and it allows you to provision a new virtual machine with a pre-configured list of software very easily.

Vagrant has completely changed how we work at Bright March. Previously, using a combination of Homebrew and manually compiling software, I would configure my computer and then try to get Neil, my co-founder, to do the same — usually with dismal results. Some library or binary would always end up missing or incorrect.

However, with Vagrant, I was able to write a shell script that completely provisioned a virtual machine for us. That script is stored as part of the configuration files for the project we are working on. With a single command, we can both launch an identical virtual machine ready for development.

Configuring the Host

Before we start setting up Symfony and Vagrant, we need to ensure that your host machine is configured properly. Fortunately, this is simple and only requires that you have a very basic version of PHP 5.5 installed on your host machine.

Windows

If you are running Windows, you can install PHP 5.5 using the installer from *http://windows.php.net/download/#php-5.5*.

Linux

If you are running a Linux distribution, you may have to manually compile PHP 5.5 because it is not available in any package manager yet. Fortunately, you only need a basic set of modules: `openssl`, `curl`, `simplexml`, `xml`, and `phar`.

The following commands will install PHP 5.5 from source on Ubuntu Linux. They assume you have already downloaded and extracted the PHP 5.5 archive and are located in its root directory as the `root` user.

```
$ apt-get update
$ apt-get install -y build-essential libssl-dev openssl \
    curl libxml2-utils libxml2 libxml2-dev libxslt1-dev \
    libcurl4-openssl-dev libmcrypt4 libmcrypt-dev

$ ./configure --with-openssl --with-zlib --with-curl \
    --enable-zip --with-xmlrpc --enable-sockets \
    --with-mcrypt --enable-mbstring --with-libxml-dir

$ make && make install
```

Mac OS X

The easiest way to install PHP 5.5 on OS X is to use the wonderful tool Homebrew. If you do not already have Homebrew installed, visit *http://brew.sh* and install it using the directions on the website. After you have it installed, you can install PHP 5.5.

```
$ brew update
$ brew tap josegonzalez/homebrew-php
$ brew install php55 php55-http php55-mcrypt
```

> *The Homebrew maintainers took the PHP formula out of the main Homebrew repositories, which is why you must tap `josegonzalez/homebrew-php` first.*

Installing Vagrant

Installing Vagrant is a very easy and straightforward process regardless of your environment. However, before you install Vagrant you will need to install VirtualBox first. VirtualBox is a virtual machine hypervisor; it provisions and manages the actual virtual machines. Many people use VirtualBox from a GUI and do not realize the GUI is simply a nice wrapper for several command line utilities.

Vagrant takes advantage of these utilities and provides a nice command line interface for them.

To install VirtualBox, navigate to *https://www.virtualbox.org* and install the binary appropriate for your host system.

Installing Vagrant is equally as simple as installing VirtualBox. To install Vagrant, navigate to *http://downloads.vagrantup.com* and install the latest available version for your operating system. This book will use Vagrant 1.2.7.

Great, now you are ready to install Symfony.

Installing Symfony

The best and easiest way to install Symfony is through Composer. Composer gives you fine-grained control over what packages are installed. Composer is a relatively new package manager for PHP. Unquestionably, it is one of the tools PHP needed to be competitive with Ruby and Python (which both have great package managers). Like many things with PHP, Composer package files are a bit more verbose than Ruby or Python package manager files, but they also give you a lot more control.

Composer uses a `composer.json` and `composer.lock` file to indicate what packages and what versions of those packages your application uses. Unsurprisingly, `composer.json` is a JSON file that describes your project.

To install Symfony, we must first install Composer. You installed PHP to ensure that Composer would work on your host system. After we install Composer and Symfony, we will have Vagrant take over and do the rest of the project configuration on the virtual machine.

> *Keeping all of my projects organized on my host machine is important to me. I generally create a `Sites` directory in my home directory and put my projects in there.*

Installing Composer

Composer is simple to install. The Composer installation, by default, gives you a nicely packaged binary file, `composer.phar`, to work with. First, Navigate to *http://getcomposer.org/download/* and run the command to install the `composer.phar` file in the root of the directory you are going to store your projects in. For example, I installed `composer.phar` in my `~/Sites/` directory.

```
#!/usr/bin/env php
All settings correct for using Composer
Downloading...

Composer successfully installed to: /Users/vcherubini/Sites/composer.phar
Use it: php composer.phar
```

As the installation suggests, run `php composer.phar` to ensure that Composer was installed successfully.

```
$ php composer.phar

   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
Composer version ef072ff8c008f35d90fe3460608bdb1365a8d7a7

...
```

Installing Symfony

You are now ready to install Symfony. Assuming you are in the directory you store your projects in, run the following command to install Symfony.

```
php composer.phar create-project symfony/framework-standard-edition MajorAuth 2.3.3
```

Composer will create a directory named `MajorAuth` and install Symfony in there. During the installation process, you may be asked to enter some application settings and database credentials. You can simply press Enter to ignore these as we will not be using that method to manage Symfony parameters at all.

```
Creating the "app/config/parameters.yml" file.
Some parameters are missing. Please provide them.
database_driver (pdo_mysql):
database_host (127.0.0.1):
database_port (null):
database_name (symfony):
database_user (root):
database_password (null):
mailer_transport (smtp):
mailer_host (127.0.0.1):
mailer_user (null):
mailer_password (null):
locale (en):
secret (ThisTokenIsNotSoSecretChangeIt):
```

Navigate into the directory you installed Symfony in, and test that Symfony was installed properly by running the command `php app/console`.

```
$ php app/console

Symfony version 2.3.3 - app/dev/debug

Usage:
  [options] command [arguments]

...
```

Great! Symfony is now installed properly.

Removing the Cruft

Now that Symfony is properly installed, it is time to clean it up and get it ready to work on. These steps could certainly be scripted, but we will go through them one by one so that you will understand exactly what is happening.

The very first thing we want to do is get everything managed by source control. The examples shown will use Git, but migrating them to your favorite version control system should be pretty trivial. To start, we want to ignore some files so they are never commited to Git.

Open a `.gitignore` file in the root of your project, and add the following lines.

```
/web/bundles/
/app/bootstrap.php.cache
/app/cache/*
/app/config/build.settings
/app/config/parameters.yml
/app/logs/*
/app/SymfonyRequirements.php
!app/cache/.gitkeep
!app/logs/.gitkeep
/build/
/bin/
/config/phing
/log/
/vendor/
nohup.out
composer.phar
phpunit.phar
REVISION
tmp
*.sass-cache*
*.swp
.DS_Store
.vagrant
```

> *We will eventually cover why all of these files are ignored.*

It is essential that you ignore the `vendor` directory. This directory is created by Composer and contains all of the code that makes up Symfony and all of the libraries you use, so you definitely do not want it committed to your project. Additionally, we ignore a lot of dotfiles that will not be used or are created by the operating system, and finally we ignore the `parameters.yml` and `build.settings` files which will hold production credentials and run-time parameters. We definitely do not want to ever commit any credentials, API keys, or configuration data to source control!

In addition to being a security vulnerability (if your code is stolen or lost, you could potentially expose your production settings), it makes working on a team more difficult. While we will do our best to normalize the development environments using Vagrant, it is still best to allow for all configuration values to easily change if necessary. This makes updating your code to work with new services easy as well.

Commit the `.gitignore` file, and then add the rest of the files not covered under `.gitignore` to your project. The `src` directory will contain the bulk of your files, which makes sense: it is where your Bundles and code are stored.

Next, you can remove several files that are not necessary for your project.

- `.travis.yml`
- `LICENSE`
- `README.md`
- `UPGRADE-2.2.md`
- `UPGRADE-2.3.md`
- `UPGRADE.md`
- `app/config/parameters.yml.dist`

Removing Incenteev

The final administration task we will want to do is to remove the Incenteev library that Symfony 2.3 installs by default. This is the library that asks for your input when installing a new version of Symfony. We will use Phing to build our `parameters.yml` file from a template and settings file, and thus do not want any interaction while building our development (and eventually production) environment.

To completely remove the Incenteev library, open up the `composer.json` file and remove the `requires` line that includes this library.

```
"incenteev/composer-parameter-handler": "~2.0"
```

> *Don't forget to remove the trailing comma after* `"sensio/generator-bundle": "2.3.*"` *so that your JSON file remains valid.*

Next, remove the lines in both the `post-install-cmd` and `post-update-cmd` sections that run Incenteev after Composer is finished updating.

```
"Incenteev\\ParameterHandler\\ScriptHandler::buildParameters",
```

And finally, remove the complete `incenteev-parameters` block in the `extra` section.

```
"incenteev-parameters": {
    "file": "app/config/parameters.yml"
},
```

Your final `composer.json` file should look like the file outlined below.

```
https://gist.github.com/leftnode/6367991
```

Now that Incenteev is removed from the `composer.json` file, you will need to update your lock file so it is not re-installed the next time you build your application.

```
$ cp ../composer.phar .
$ php composer.phar update

...
```

> *You will notice I first copied the `composer.phar` file to the project root. This simply makes it easier to locate and execute.*

After Composer has updated the lock file, commit both `composer.json` and `composer.lock` to your Git repository.

Bootstrapping With Vagrant

You are now ready to use Vagrant to bootstrap your development environment. To keep things simple, we will use a 64bit version of Ubuntu 12.04 LTS. Vagrant does a wonderful job of providing a pre-built virtual machine image for you, and installing it is extremely simple. LTS — long term support — versions of Ubuntu are preferred for server operating systems because they are officially supported for five years by Canonical.

The three commands you will use most often with Vagrant are: `vagrant up`, `vagrant ssh`, and `vagrant halt`. The command `vagrant up` provisions your virtual machine and brings it up. `vagrant ssh` allows you to SSH into your virtual machine. Finally, `vagrant halt` completely halts and shuts down your virtual machine, but leaves its hard-disk in-tact for when you bring the virtual machine back up.

Vagrant provides other commands to destroy, suspend, and resume your virtual machine. I have found the three outlined above to be the most useful.

In the root of your Symfony installation, run this command to install the virtual machine from Vagrant. You will only have to run this command one time. After you run this command, you can use Vagrant in a totally separate project with the same command — simply leave out the URL to the virtual machine image.

```
vagrant init precise64 http://files.vagrantup.com/precise64.box
```

> *The codename for Ubuntu 12.04 is Precise Pangolin, which is why the virtual machine image is named* `precise64` *— Precise Pangolin 64bit.*

Because this is the first time you ran Vagrant, the actual virtual machine image will be downloaded for you. This may take a few minutes.

Vagrant can intelligently determine where the `Vagrantfile` is located in your project, but it is best to execute all of the Vagrant commands in the same directory it is located in.

`Vagrantfile`

After the virtual machine is downloaded and installed, Vagrant will create a new file named `Vagrantfile` in the root directory of your project. This is a configuration script written in Ruby and tells Vagrant how to provision and build your virtual machine. The default `Vagrantfile` is made up almost entirely of comments.

I generally keep the comments in the `Vagrantfile` for each of my projects, but you are welcome to remove them. I have removed most of the comments for display purposes in this book. However, they are maintained in the actual source code that accompanies the book itself.

Below is the entirety of the `Vagrantfile` used for building MajorAuth. We will take the time to go through line by line to explain how everything works.

```
# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = "precise64"
  config.vm.provision :shell, :path => "app/config/vagrant-bootstrap.sh"

  # Create a forwarded port mapping.
  config.vm.network :forwarded_port, guest: 8000, host: 8000, auto_correct: true

  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # Ignore for Windows.
  config.vm.network :private_network, ip: "192.168.100.100"

  # Share an additional folder to the guest VM.
  # Remove , :nfs => true if using Windows.
  config.vm.synced_folder ".", "/var/apps/major-auth", :nfs => true

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
    v.customize ["modifyvm", :id, "--cpus", "8"]
    v.customize ["modifyvm", :id, "--cpuexecutioncap", "100"]
    v.customize ["modifyvm", :id, "--hwvirtex", "on"]
  end
end
```

First, we begin by telling Vagrant to use the `precise64` virtual machine image previously installed. Next, we tell Vagrant how to provision this virtual machine by pointing to a shell script we will write. This shell script will bootstrap the entire environment necessary for developing a Symfony application. Vagrant supports using Chef and Puppet to provision your virtual machine as well, however, I highly recommend you learn how to build it from scratch (from source), and then move on to more abstract topics like Chef or Puppet.

Forwarding a port is important so you can use the built-in PHP server for development. While it should never be used in production, Symfony works great with it for development. By default, it runs on port 8000 so we forward that port to our host machine.

One incredibly useful feature of Vagrant is its ability to manage multiple synced directories against the host and guest operating systems. These directories are transparent, meaning if you create a file while on the virtual machine, the file will exist on the host machine as well.

Symfony (and Doctrine) aggressively cache a lot of files during development. Caching all of these files and syncing them with the VM can take a lot of time — sometimes several seconds on slower host machines. This means each request to the virtual machine can take two to four seconds, which makes development agonizing. To get around this issue, we use NFS — network file system — to sync the files.

To use NFS, you must assign a static IP to the virtual machine. You may have to change the IP address above to work with your home or work router. Additionally, Vagrant will require sudo access when provisioning your virtual machine so be prepared to enter your password.

Now that Vagrant knows to assign a static IP to the virtual machine, we also want to automatically create a synced directory when the virtual machine is provisioned. Vagrant will use NFS to sync files to this directory as well.

And finally, we tell Vagrant to pass some additional command line parameters to the headless version of VirtualBox.

- --**memory**: Set the memory to 1024MB.
- --**cpus**: Give the virtual machine eight processors.
- --**cpuexecutioncap**: Allow the virtual machine CPUs to use 100% of the available resources if necessary.
- --**hwvirtex**: Enables "hardware virtualization extensions" which will use specific extensions built into your hardware to speed up the virtual machine.

> *These are my default settings for my 8-core 16GB RAM MacBook Pro. Obviously, if your machine is less powerful than mine, you will need to adjust these settings.*

`vagrant-bootstrap.sh`

Before you can fully provision your virtual machine, we need to tell Vagrant how to build it. This is where the `vagrant-bootstrap.sh` file comes into play.

As previously mentioned, this is a shell script that builds your virtual machine from scratch. It installs the necessary libraries needed, and then goes on to compile Ruby, PostgreSQL, Redis, and PHP from source.

> *My general philosophy of server administration is that you install libraries from your Linux distributions package manager and compile your vital applications from source. For example, `libxml2` is installed from Aptitude, but PHP is compiled from source. Essential libraries are more likely to be very stable, so trusting the maintainers is easy. By installing essential binaries from source, you have greater control over the upgrade paths on your production systems.*

The `vagrant-boostrap.sh` file is long, so it will be presented in its entirety and then we will disect it.

```
https://gist.github.com/leftnode/6370185
```

Understanding how this file works is important because it will be the basis for how we provision our production server. The `vagrant-bootstrap.sh` file should be placed in the `app/config/` directory as specified in the `Vagrantfile`.

No Redundant Provisioning

The very first thing we want to check when bringing up a virtual machine is if it has already been provisioned. If it is, we do not want to provision it again. As the comment states, you can run the command `vagrant up --no-provision` to bring up the virtual machine without provisioning it. However, if you leave off the `--no-provision` argument flag, the machine will be provisioned. Thus, this initial check prevents that from happening.

Databases

During development, we will be using two databases: `major_auth` and `major_auth_test`, each with their own eponymous user. Creating two separate databases makes sense: the primary database, `major_auth`, will be used for browser and blackbox testing whereas the `major_auth_test` database will be used for automated testing.

The `major_auth_test` database will have its data deleted and re-inserted many times throughout the course of a test-suite run. The `major_auth` database will have its data manipulated when actually using the application as a normal user would.

Default Packages

The lines beginning with `apt-get` install essential software needed to compile Ruby, PostgreSQL, Redis, and PHP. These lines install the libraries through the package manager.

Date and Time Configuration

We want to ensure that the server is in the UTC timezone. Our production server will use UTC, so we want our development environment to mimic that. This will almost certainly be a different timezone than your host computer. Additionally, we set the default language and locale as English, UTF-8.

Compilation

The `/etc/skel/.profile` file on Ubuntu servers stores the default `.profile` file added to a users home directory when it is created. I use a customized `.profile` file to add the default PostgreSQL binary location to my `$PATH`.

> *You will notice all software and files are downloaded from an Amazon S3 bucket named `build.brightmarch`. I store everything in that bucket because not all vendors provide an easily discoverable direct link to their tarballs (PHP) and some FTP servers are very, very slow (Ruby). The files uploaded are identical to the ones provided by the vendors. You are free to use my bucket and bandwidth.*

We can finally start compiling all of the software necessary for the server. First, we create all of the directories in `/opt/src/` to store all of the source code.

Ruby

Compiling Ruby is simple. Although we are building a PHP application, we will make extensive use of Ruby software. Installing Ruby 2.0.0 will also install the `gem` program which makes installing Ruby applications simple.

After we compile Ruby, four pieces of software are installed: `sass`, `compass`, `god`, and `zurb-foundation`.

Sass and Compass are used to manage CSS. We will not be covering them in depth, but they are useful tools we will use alongside ZURB Foundation. Foundation is a CSS framework that makes it incredibly simple to build beautiful, well constructed applications. It uses Sass as well.

The application `god` is a "process monitoring framework". Essentially, we will use it to monitor our Resque workers to ensure they always stay up. You can read more about `god` at *http://godrb.com* and we will cover it in depth when we begin talking about Resque and background workers.

PostgreSQL

Compiling PostgreSQL is surprisingly simple. We start by creating a user, `postgres`, that will run all of the PostgreSQL services. The home directory for the `postgres` user is also where we will store all of the database data (PostgreSQL calls this data the cluster).

After PostgreSQL is compiled and installed, we run a command as the `postgres` user to initialize the database cluster in `/home/postgres/cluster/`. Additionally, we tell PostgreSQL the default encoding for the databases will be UTF-8.

The last piece of administrative work necessary is to install the PostgreSQL initialization script. Ubuntu places these scripts, called services, in `/etc/init.d/`. The initialization script is downloaded and installed during the provisioning process. This ensures that Ubuntu starts PostgreSQL when Vagrant starts the server.

Now that PostgreSQL is installed and running, we can create the default databases and users.

> *PostgreSQL will also refer to users as roles.*

Redis

Redis is also incredibly simple to install. Like PostgreSQL, we want Redis to start when the server starts, so an initialization script is installed as well. Additionally, we have to tell Redis to run in the background as a daemon because it does not by default.

PHP

Finally, PHP is ready to be compiled and installed. To keep this instance of PHP as light as possible, only the most essential extensions are compiled.

By default, PHP looks in `/usr/local/lib/` for the `php.ini` file, so we copy the development sample file there.

We want to add the `redis` extension in addition to the extensions we compiled into PHP. This is extension is not bundled with PHP by default.

To complete the installation of PHP, we add the `redis` extension to the `php.ini` file and tell PHP the default timezone is UTC.

Finishing Up

We're almost there! The last few steps are rather simple. By default, Vagrant creates a user named `vagrant` which is the user you use to SSH into the virtual machine. First, we ensure that the `.profile` file downloaded earlier is in our home directory *on the virtual machine*. Next, several pre-written files are downloaded. These are bash and Vim files that I have built up over the years to make command line work more enjoyable. Of course, you're free to replace them with the files that make your development environment the most comfortable.

Finally, we create the file that lets Vagrant know the box is fully provisioned.

And with that, your Vagrant virtual machine is fully provisioned.

Your Vagrant

Now that your Vagrant virtual machine is provisioned, you can access it using the command `vagrant ssh`. This command is identical to the `ssh` command you are already familiar with, with the exception that you can only access your virtual machine with it.

If you will recall, we told Vagrant to sync the directory `/var/apps/major-auth` to the root of our project. Navigate to that directory and create a sample file there.

```
$ cd /var/apps/major-auth
$ echo "In Your Vagrant" > in-vagrant.txt
```

In another terminal session, navigate to your project directory *on your host machine*. You will see the file `in-vagrant.txt` listed along with the contents *In Your Vagrant*.

File syncing is obviously very powerful and will allow us to develop on our host machine while having the virtual machine execute all of our code.

> *I have included a helper alias in the `.bash_aliases` file. You can run the command `goapps` and it will take you directly to the `/var/apps/` directory.*

Because your virtual machine is quite literally a Linux server, you can simply use the `exit` command to exit it and return to your host machine.

In This Chapter

In this chapter, you learned how to compile the essential architecture of your development server (a skill that will be used when provisioning your production server), and how to use Vagrant to manage your virtual machines. Additionally, you installed Symfony, tracked it in Git, and cleaned up some of the cruft that naturally comes with a default installation.

You are now ready to begin laying out your Symfony application.