['ruby', 'javascript', 'python']

development environments for beginners

seth vincent

Ruby, Javascript, & Python: Development Environments for Beginners

Get started with ruby, python, and javascript development environments

Seth Vincent

This book is for sale at http://leanpub.com/dev-envs

This version was published on 2014-02-28

Also By Seth Vincent

Learn.js #1

the moon box

Learn.js #2

Learn.js #3

Theming with Ghost

npm recipes

Contents

1 Hello, dear reader.

Thank you

Thank you for buying this book! It is independently published, and each sale makes a significant difference.

Why write this?

When I first started programming, learning the languages was the easy part compared to figuring out text editors, version control, testing, and all the different tools that come along with writing code for a real project.

My goal for this book is to help flatten the learning curve a little, so that going from hello world to a complex application isn't quite as difficult for you as it was for me when I started out.

About the book

If you're not sure how to choose between programming in ruby, python, or javascript, this guide will get you familiar with the tools, syntax, and workflow associated with each language.

By comparing ruby, python, and javascript development environments you'll get a strong sense of which language best fits your style.

Maybe you'll learn that you want to work with all three of these languages, or two out of three. Whatever your decision, you'll be able to use this guide as a reference if you need help remembering how to set up a development environment for any one of these three languages.

Main concepts you'll learn:

- command-line tools
- virtual machines
- text editors
- version control
- package managers
- documentation
- testing
- language basics
- intro to web frameworks

This book is open source

Contribute errata or content requests at the GitHub repository for this book: github.com/sethvincent/dev-envs-book[1].

Other books

There's a good chance that if you like this book you'll be interested in the other books in the Learn.js series.

Check them out!

- Introduction to JavaScript & Node.js[2]
- Making 2d Games with Node.js & Browserify[3]
- Mapping with Leaflet.js[4]
- Theming with Ghost[5]
- npm recipes[6]

Learn more at learnjs.io[7].

[1] https://github.com/sethvincent/dev-envs-book
[2] http://learnjs.io/books/learnjs-01
[3] http://learnjs.io/books/learnjs-02
[4] http://learnjs.io/books/learnjs-03
[5] http://themingwithghost.com
[6] http://learnjs.io/npm-recipes
[7] http://learnjs.io

2 What is a development environment?

Wait, what is this development environment thing?

A development environment is like a workshop. A space and set of tools for building projects before they are released into the world wide web. Everything you need to build your project should exist in your development environment, and ideally your development environment will be similar or identical to the production environment on which your project will be released.

A development environment is a stage in the workflow, or release cycle, of a project.

Other stages can include:

- testing
- staging
- production

There can be other stages, too, depending on the complexity of the project and the requirements you must meet for quality assurance and user testing.

In this book we take a look at setting up development environments for Ruby, JavaScript, and Python, and introduce the basics of each of those three languages.

3 Vagrant: install an operating system inside of your operating system!

Vagrant is a tool for running "virtual machines" on a computer. Let's say your computer is running the Windows operating system. With vagrant, you can run a virtual machine on your computer that is running another operating system, like Ubuntu.

It's almost like dual booting, like having both Windows and Ubuntu installed on a computer, except you have a lot more control over a virtual machine. You can spin one up temporarily to work on a project, and when you're done, destroy the virtual machine.

Vagrant makes this process easy. It relies on virtual machine software – there are a number of options, but we'll use software called virtualbox.

In a way, vagrant is a wrapper around virtualbox – making it easy to create, run, and destroy virtual machines that run from the command line.

The primary operating system on your computer is called the "host" operating system. Any virtual machines you create are called "guest" operating systems.

We'll use the term **box** to describe the virtual machines that we create using vagrant.

Reasons for using vagrant:

- Keep your host OS clean by installing dependencies for your project in a virtual machine rather than having everything installed on your host OS
- Have the same operating system and same dependencies set up in your virtual machine as you have on the production server, making it easier to deploy your application because there's little difference between the two environments.
- The people on your team can use vagrant so that all of your development environments match, easing issues with some people having issues with developing on a particular operating system.

Vagrantfiles

Vagrant uses a file named `Vagrantfile` in your project directory as a config file for your vagrant box.

There are many config options you can set. To learn more check out the Vagrantfile section of the vagrant documentation: http://docs.vagrantup.com/v2/vagrantfile/index.html[1].

[1] http://docs.vagrantup.com/v2/vagrantfile/index.html

Website

http://www.vagrantup.com/

Install

Virtualbox:

First we install virtualbox[2].

Go to the virtualbox downloads folder: https://www.virtualbox.org/wiki/Downloads[3].

Click the link for your operating system to download the virtualbox installer. Once it's finished downloading, run the installer.

If needed, you can follow the **virtualbox installation instructions**: http://www.virtualbox.org/manual/ch02.html[4]

Documentation

You probably don't need to, but if you want to dig into virtualbox in depth, check out the documentation: https://www.virtualbox.org/wiki/Documentation[5].

Vagrant

Installing vagrant is pretty easy. Just grab the downloader for your operating system from the vagrant downloads page:

download vagrant:

http://www.vagrantup.com/downloads[6]

Choose the installer package for your operating system.

installation instructions:

docs.vagrantup.com/v2/installation[7]

Documentation

docs.vagrantup.com/v2[8]

[2] https://www.virtualbox.org/
[3] https://www.virtualbox.org/wiki/Downloads
[4] http://www.virtualbox.org/manual/ch02.html
[5] https://www.virtualbox.org/wiki/Documentation
[6] http://www.vagrantup.com/downloads
[7] http://docs.vagrantup.com/v2/installation/index.html
[8] http://docs.vagrantup.com/v2/

Set up your first vagrant machine

Let's get started with the basics of using vagrant.

Open the terminal on your computer and run this command:

```
vagrant
```

Running the command by itself will show you all the possible sub-commands and options you can pass.

Let's try this thing out.

Create and navigate to your `dev-envs` folder:

```
mkdir ~/dev-envs
cd ~/dev-envs
```

Create a folder named `tmp` and change directory into it:

```
mkdir tmp && cd tmp
```

Run `vagrant init` to create a Vagrantfile in our tmp directory:

```
vagrant init precise32 http://files.vagrantup.com/precise32.box
```

By passing the name `precise32` and the url of the box we want to use, we prepopulate the Vagrantfile with the vagrant box we intend to use for our project. And by passing a url we're letting vagrant know that we want to download the vagrant box, as it isn't on our computer yet.

precise32 is the name for Ubuntu 12.04 LTS 32-bit.

You should see output on the terminal like this:

```
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.
```

Now, run `vagrant up` to boot your vagrant box:

```
vagrant up
```

This does a couple things: because we passed a url to vagrant init, and we don't already have a box on our machine named precise32, `vagrant up` will first download the precise32 box, then boot it with any configuration that's been set in the Vagrantfile.

You should see output on the terminal like this:

```
Bringing machine 'default' up with 'virtualbox' provider...
[default] Box 'precise32' was not found. Fetching box from specified URL for
the provider 'virtualbox'. Note that if the URL does not have
a box for this provider, you should interrupt Vagrant now and add
the box yourself. Otherwise Vagrant will attempt to download the
full box prior to discovering this error.
Downloading or copying the box...
Extracting box...te: 1727k/s, Estimated time remaining: 0:00:01)
Successfully added box 'precise32' with provider 'virtualbox'!
[default] Importing base box 'precise32'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Fixed port collision for 22 => 2222. Now on port 2201.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2201 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

Now that the box is up and running, we can ssh into this instance of Ubuntu that we just set up!

We do that by running `vagrant ssh` in the terminal:

```
vagrant ssh
```

You should see output on the terminal like this:

```
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:22:31 2012 from 10.0.2.2
```

You have now entered your vagrant box. Every command you type in your terminal now happens in this instance of Ubuntu Linux.

To exit from the vagrant box, and take the terminal back to your host operating system, run the `exit` command:

```
exit
```

You prompt should now look the same as before you ran `vagrant ssh`.

Now that we downloaded that precise32 box once, we won't have to do it again. We'll be able to use it for each new project we start.

Next time we're about to create a vagrant box we'll navigate to the project folder and run 'vagrant init' again, like this:

```
vagrant init precise32
```

We don't have to pass in the url, because we've already downloaded the box.

When we run `vagrant up`, the box won't be downloaded, because we've already got a copy of it sitting on our computer. This speeds things up.

To see which boxes you've currently got downloaded run this command:

```
vagrant box list
```

Let's stop this vagrant instance we created in our ~/dev-envs/tmp folder.

Run this command:

```
vagrant halt
```

We can start the box back up again any time by running `vagrant up` from inside this folder.

Let's now destroy the vagrant instance.

To learn about this use the following command:

```
vagrant destroy --help
```

You'll see output like this:

```
Usage: vagrant destroy [vm-name]

    -f, --force                      Destroy without confirmation.
    -h, --help                       Print this help
```

Alternatives to vagrant/virtualbox

docker

Docker is a cool new approach that uses lightweight containers for encapsulating applications. Definitely worth checking out: http://www.docker.io[9].

Go through their getting started tutorial to get a sense of how it works: http://www.docker.io/gettingstarted[10].

It's also possible to use docker on one of your vagrant machines, which seems like an awesome option. If you're feeling adventurous, check out docker's documentation for integrating with vagrant and virtualbox: http://docs.docker.io/en/latest/installation/vagrant/[11].

nitrous.io

You can have a development environment that you access through the web using nitrous.io: https://www.nitrous.io[12].

This is great if you're on a machine that runs an operating system like ChromeOS, or it's something you can't install software on for whatever reason.

You can set up one box on nitrous.io for free, and beyond that it costs money.

[9] http://www.docker.io/
[10] http://www.docker.io/gettingstarted/
[11] http://docs.docker.io/en/latest/installation/vagrant/
[12] https://www.nitrous.io/

4 Terminal: conquer the command line

Get excited, it's time to use the terminal.

The most important thing to keep in mind: **don't be afraid of the terminal**.

You can only break your computer using the terminal if you do really weird stuff. I mean, you mostly have to go out of your way to break your computer. There are a few ways you can crunch your machine, so it's worthwhile to be skeptical about new commands and research what they do before you use them.

Each command that you run on the terminal will use a pattern similar to this:

```
name-of-command --options input-file-or-text output
```

The `name-of-command` is the actual command. Options are often preceeded by two dashes, or they can likely be shorted to one dash and the first letter or an abbreviation. Then, there will occasionally be some kind of input text or file that the command is acting on, or changing. Similarly, you might specify a filename for the output of the command. You'll see that many of the commands below are more simple.

Vagrant

Because we'll be using vagrant to create a virtual machine running Ubuntu throughout this book, this chapter will teach usage of Linux / Unix terminal commands.

Vagrant

Let's create a vagrant machine in your javascript dev-envs folder:

```
mkdir ~/dev-envs/terminal
cd ~/dev-envs/terminal
```

Create a new vagrant machine using the Ubuntu Precise box:

```
vagrant init precise32
```

Now start the vagrant machine:

```
vagrant up
```

If all goes well that'll result in output similar to the following:

```
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'precise32'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Fixed port collision for 22 => 2222. Now on port 2200.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2200 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

Now we will log in to the vagrant machine. This will be very much like using the `ssh` command to log in to a remote server.

Use this command:

```
vagrant ssh
```

You should see output similar to the following:

```
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:22:31 2012 from 10.0.2.2
```

Now we can experiment with terminal commands

Basic commands:

Changing directory:

```
cd path/of/directories
```

`cd` on its own, or `cd ~` will take you to your home directory.

Create a directory:

```
mkdir directory-name
```

Show the directory you're currently in:

```
pwd
```

List the files in the directory:

```
ls
```

List the files in a more readable way, with useful information like permissions included:

```
ls -al
```

Open a file with its default application:

```
open filename
```

Open current directory in the finder:

```
open .
```

or open some other directory:

```
open path/to/directory
```

Create an empty file if it doesn't already exist:

```
touch file-name
```

Open and edit a file with the simple nano editor:

```
nano file-name
```

Move a file or directory:

```
mv file new/path/to/file
```

Rename a file or directory:

```
mv file new-file-name
```

Copy a file:

```
cp file name-of-file-copy
```

Copy a directory

```
cp -R directory directory-copy
```

The `-R` option allows for recursive copying of files inside the directory.

Delete a file:

```
rm file-name
```

Delete a directory and its contents:

```
rm -rf path/to/directory
```

Let's dissect this command:

`rm` is for deleting things.

`-rf` means that files will be recursively deleted, and the deletion will be forced. `r` is for recursive, `f` is for forced.

Never do this:

```
rm -rf /
```

Be very careful with the rm command. You can easily delete things on accident.

This command is deleting with the `rm` command, recursively forcing the deletion of files and folders with `-rf`, and we've passed the root directory, `/` as the thing to delete. This means it will delete everything on your hard drive. Some operating systems protect against this mistake, and if you're not the root user you would need to prefix this command with `sudo` to make it work. Be careful, and be nice.

Clear the terminal screen of previous activity:

```
clear
```

Reset the terminal:

```
reset
```

Stop a running process:

```
control+C
```

If a process is running in the terminal and you need to stop it, press the `control` key and the `C` key at the same time.

Run multiple commands on one line:

```
&&
```

With `&&` you can chain together multiple commands that execute one after the other. This example creates a directory, then moves you into that new directory:

```
mkdir new-folder && cd new-folder
```

Aliases and environment variables

Aliases

Aliases allow you to create abbreviated commands that alias long, complex, or regularly used commands.

Here is an example:

```
alias l="ls -al"
```

The above aliases the `ls -al` command to a shortened `l`.

To create an alias you will open the .bashrc file in your home folder.

Open the .bashrc file with nano:

```
nano ~/.bashrc
```

Add the following alias to the bottom of the file:

```
alias pizza="echo 'pizza is awesome!'"
```

Save the file by pressing `control + O`.

Exit nano by pressing `control + x`.

Environment variables

Environment variables represent values that are useful across for processes running on your computer.

Reading an environment variable:

In the terminal, run the following:

```
echo $HOME
```

If you're logged into a vagrant machine, you'll see output like this:

```
/home/vagrant
```

This is your home folder, also known as your user folder.

On a Mac you'll see output like this:

```
/Users/your-user-name
```

Setting an environment variable

In the terminal, set an environment variable like this:

```
PIZZA="oooooh, pizza"
```

Now, you can read the variable the same as we did before:

```
echo $PIZZA
```

If you close or reset your terminal session, you'll lose this temporary variable. To save an environment variable so it can be accessed in all your sessions, we'll place the definition of the variable in the ~/.bashrc file.

Open the ~/.bashrc file:

```
nano ~/.bashrc
```

Add the following to the bottom of the ~/.bashrc file:

```
export PIZZA="oooooh, pizza"
```

Source the .bashrc file:

```
source ~/.bashrc
```

Now, you can close the terminal window, open a new one, and run the following command:

```
echo $PIZZA
```

And you'll still see the following output:

```
oooooh, pizza
```

5 Text editors

Sublime Text Editor

Sublime is a popular text editor with versions for Mac, Windows, and Linux.

You can download an evaluation copy for free, and pay for a license when you're ready.

In this book we'll be using version 2 of Sublime, in future updates to the book we'll switch to version 3.

Install

Go to sublimetext.com[1] and click the download button.

This will download a .dmg file. Once the download has completed, double-click the .dmg file. Next, drag the Sublime Text application into your Applications folder.

Use Sublime from the command line

To use Sublime from the command line using Mac or Linux, you'll need to create a symbolic link:

```
1 ln -s "/Applications/Sublime Text 2.app/Contents/SharedSupport/bin/subl" ~/bin/su\
2 bl
```

If you get an error you may need to create the `bin` folder:

```
1 mkdir ~/bin
```

`ln -s` is a command used for creating symbolic links, which you can think of like aliases.

The first argument is the location of the original file, the second argument is the new, alias location.

Now, you can run the `subl` command on the terminal.

When you run `subl --help`, you'll get help information for the command that looks like this:

[1] http://www.sublimetext.com/

```
Usage: subl [arguments] [files]         edit the given files
   or: subl [arguments] [directories]   open the given directories
   or: subl [arguments] -               edit stdin

Arguments:
  --project <project>: Load the given project
  --command <command>: Run the given command
  -n or --new-window:  Open a new window
  -a or --add:         Add folders to the current window
  -w or --wait:        Wait for the files to be closed before returning
  -b or --background:  Don't activate the application
  -s or --stay:        Keep the application activated after closing the file
  -h or --help:        Show help (this message) and exit
  -v or --version:     Show version and exit

--wait is implied if reading from stdin. Use --stay to not switch back
to the terminal when a file is closed (only relevant if waiting for a file).

Filenames may be given a :line or :line:column suffix to open at a specific
location.
```

Tips for using Sublime

Searching files

Basic Search To do a basic text search, press `Command + f` to open the search bar.

Here you can choose to use regular expressions or simple string searches.

You can cycle through instances of a search phrase by clicking **Find** and **Find Prev**, and you can highlight all instances of a search phrase by clicking **Find All**.

Exit the search bar by pressing the `esc` key.

Search all project files To search all files in your project, press `Shift + Command + f` to open the search panel.

After you enter a search phrase in the **Find** input, press enter or click **Find**. A new tab will then open with all relevant search results.

In the **Where** input you can specify files and folders that you want to search. Click the `...` button to the right of the input field to use a GUI to add and remove files and folders.

Optionally you can use the **Replace** input to specify a phrase to replace the search phrase.

To exit this search panel press the `esc` key.

Finding and switching between files Use the `Command + P` keyboard shortcut to bring up a menu that allows you to quickly navigate to other files in the project. Start typing the name of the file to filter the results.

Finding specific methods or functions Use the `Command + R` keyboard shortcut to get a menu that allows you to find classes, methods, functions, variable declarations, and other important pieces of code. Like the file menu mentioned above, you can start typing a name to filter the results.

Jumping to a specific line number There are two ways: `Control + G`, enter the line number, and hit return. Or `Command + P`, type a colon, then the line number, then hit return.

Fun with multiple cursors

One of the most enjoyable features of Sublime (and similar text editors) is being able to use multiple cursors for quickly editing multiple parts of a text file.

Search and Find All One way to get multiple cursors in a file is by searching using `Command + f`, entering a search phrase, and clicking **Find All**. This will heighlight all instances of the search phrase. Next you can press the left and right arrows to navigate around the text and make revisions like normal, only you'll be editing the text in multiple places.

Selecting multiple instances of a word with `Command + d` An even faster way of selecting multiple instances of a word is by clicking a word, then pressing `Command + d`. Pressing is once will select the word that the cursor was on. Pressing it repeatedly will select the next instance of the word in the text document, and eventually wrap around to the top of the file until it has searched up to the original word you clicked on. Hold down `Command + d` to quickly select all instances of the word the cursor is on.

A shortcut to selecting all instances of a word at once is through the use of `Command + Control + G` on Mac or `Alt + F3` on Windows.

Using the `option` key By holding the `option` key and dragging up and down in a straight line over rows of text you'll create a cursor on each row of text.

Indenting quickly with `Command + {` and `Command + }` Select a block of text and indent it to the left with `Command + {` and to the right with `Command + }`.

Start and end of files and lines with the `Command` and arrow keys.

By holding `Command` and using the arrow keys you can quickly move to the start and end of the file and lines of text.

`Command + Left Arrow` moves the cursor to the left side of the text line the cursor is on.

`Command + Right Arrow` moves the cursor to the right side of the text line the cursor is on.

`Command + Up Arrow` moves the cursor to the top of the text document.

`Command + Down Arrow` moves the cursor to the bottom of the text document.

By holding the `Shift` key along with any of the above four key combinations allows you to select text from the cursor's starting location to ending location of the cursor determined by the command.

This is particularly useful for selecting a line of text. Use `Command + Right Arrow` to go to the end of the line, then `Shift + Command + Left Arrow` to select that entire line of text.

Using the Sublime console

Open the console using "ctrl + ~~~~~~~~ (control plus the backtick key) or by going to **View > Show Console** in the top menu.

This is a Python console that uses Sublime's embedded version of Python intended for interacting with the Sublime plugin API.

Compatibility with TextMate

You can use TextMate snippets, color schemes, `.tmLanguage` files, and `.tmPreferences` files with Sublime 2.

Installing Sublime packages with Package Control

You can easily install third-party packages in Sublime to add new functionality by using a package manager called Package Control. This is useful for adding new color schemes, language syntax highlighters,

The Package Control website: sublime.wbond.net[2]

Installing Package Control

Install Package control by copying and pasting the following code into the Sublime console (press "Control + ~~~~~~~~ to open the console).

[2] https://sublime.wbond.net

```
import urllib2,os; pf='Package Control.sublime-package'; ipp = sublime.installed_\
packages_path(); os.makedirs( ipp ) if not os.path.exists(ipp) else None; urllib2\
.install_opener( urllib2.build_opener( urllib2.ProxyHandler( ))); open( os.path.j\
oin( ipp, pf), 'wb' ).write( urllib2.urlopen( 'http://sublime.wbond.net/' +pf.rep\
lace( ' ','%20' )).read()); print( 'Please restart Sublime Text to finish install\
ation')
```

Next you'll need to restart Sublime.

Finding and installing Packages

Resources

Website

Check out the main Sublime website: sublimetext.com[3]

Documentation

Official Sublime documentation: sublimetext.com/docs/2[4]

Sublime Text unofficial documentation: docs.sublimetext.info/en/sublime-text-2[5]

Alternative text editors:

There are too many to list. Two that I recommend learning about are vim & nano.

vim

vim has a steep learning curve, so save this for later after you've mastered other tools. To learn more about vim, check out this list of resources: net.tutsplus.com/articles/web-roundups/25-vim-tutorials-screencasts-and-resources/[6]

nano

nano is a simple, easy to use editor that you'll usually find on unix/linux operating systems.

There are a few basic key commands you need to know to start out with nano:

Edit something with nano:

[3] http://www.sublimetext.com/

[4] http://www.sublimetext.com/docs/2/

[5] http://docs.sublimetext.info/en/sublime-text-2

[6] http://net.tutsplus.com/articles/web-roundups/25-vim-tutorials-screencasts-and-resources/

```
nano filename.txt
```

Save a file:

```
control+O
```

Exit from nano (you'll be prompted to save):

```
control+X
```

Search for some text:

```
control+W
```

6 Git: it's like File > Save, only collaborative

Seriously. You know how important it is to save your work. We've all been beaten into a sad sack of anger and disappointment when we've lost our work.

Consider git to be the equivalent of File > Save that keeps track of every version of your work, and allows you to share those versions with other people and collaborate in a way that won't have you overwriting each other's changes.

Git is version control software.

There are many alternatives to git, but it has become a standard for developers in large part because of github.com, a service for hosting code using git.

The best way to start learning git (and GitHub) is to visit try.github.com[1]. You should also try githug, a game for learning git[2].

Project website:

http://git-scm.com/

Install

download / install: http://git-scm.com/downloads

If you are using a Mac, you can install using homebrew[3]:

```
1 brew install git
```

> For more information about homebrew, check out the project's homepage: brew.sh[4].

On Debian/Ubuntu, install using apt-get:

---

[1] http://try.github.com

[2] https://github.com/Gazler/githug

[3] http://brew.sh/

[4] http://brew.sh/

```
apt-get install git
```

For Windows machines, download git from the git website: git-scm.com/downloads[5]

Documentation

docs: http://git-scm.com/documentation

Basics

Here are some basics of using git:

Create a git repository:

```
cd name-of-folder
git init
```

Add files:

```
git add name-of-file

// or add all files in directory:

git add .
```

When you add files to a git repository they are "staged" and ready to be committed.

Remove files:

```
git rm name-of-file

// force removal of files:

git rm -rf name-of-file-or-directory
```

Commit files and add a message using the `-m` option:

[5] http://git-scm.com/downloads

```
git commit -m 'a message describing the commit'
```

Create a branch:

```
git branch name-of-branch
```

Checkout a branch:

```
git checkout name-of-branch
```

Shortcut for creating a new branch and checking it out:

```
git checkout -b name-of-branch
```

Merge a branch into the master branch:

```
git checkout master
git merge name-of-branch
```

Add a remote repository:

```
git remote add origin git@github.com:yourname/projectname.git
```

List associated repositories:

```
git remote -v
```

Pull changes from a remote repository:

```
git pull origin master
```

Push changes to a remote repository

```
git push origin master
```

Checkout a remote branch:

```
git checkout -t origin/haml
```

Resources

Official documentation

The official git documentation[6] has an API reference[7], a book about git[8], videos[9] instructing on basic usage of git, and a page of links[10] to useful resources.

try.github.io[11]

This interactive tutorial is a great introduction to both git and GitHub. Highly recommended.

gitready.com[12]

I often refer to gitready.com for various tips and tricks for using git.

help.github.com[13]

The help section of GitHub's site has instructions for using github.com, but it also has some great documentation for using git.

---

[6] http://git-scm.com/doc
[7] http://git-scm.com/docs
[8] http://git-scm.com/book
[9] http://git-scm.com/videos
[10] http://git-scm.com/doc/ext
[11] http://try.github.io/
[12] http://gitready.com/
[13] https://help.github.com/

7 GitHub.com: a social network for git users

GitHub is a great place to host your code. Many employers hiring for developer and designer positions will ask for a GitHub profile, and they'll use your GitHub activity as part of the criteria in their decision-making process.

In fact, if you're looking to get a job with a particular company, try to find *their* GitHub profile and start contributing to their open source projects. This will help you stand out, and they'll already know your technical abilities based on your open source contributions. That's a big win.

GitHub has become the de facto code hosting service for most open source communities.

Website

http://github.com

Create an account

If you haven't already, create an account at github.com[1].

Create your first repository

GitHub Pages

GitHub Pages is a web hosting service offered by GitHub typically used for hosting websites for open source projects that have code hosted on GitHub. In practice, people use it for all kinds of sites. The only limitation is that it is strictly for static sites composed from html, css, and javascript.

Learn more about GitHub Pages: http://pages.github.com[2]

[1] http://github.com
[2] http://pages.github.com

With GitHub Pages you can:

- design a website any way you want by having complete control over the html, css, and javascript.
- use simple templates for getting started using GitHub Pages.
- create sites for yourself and all of your projects hosted on GitHub.
- use a custom domain name if you want!

Visit the help section for GitHub Pages[3] to learn more details about hosting sites on GitHub.

Host a site for free

GitHub has a useful service called GitHub Pages[4] that allows you to host a simple site on their servers for free.

To get started, fork this simple template: github.com/maxogden/gh-pages-template[5].

Visit that github project, make sure you're logged in, and click Fork in the upper right side of the screen.

Fork gh-pages-template to your personal account.

Rename the repository from gh-pages-template to whatever you want by clicking on Settings on the right side of your fork of the repository, and changing the name there.

That's it! You now have a website hosted through GitHub Pages.

You'll be able to visit your site at **YOUR-USERNAME**.github.com/**YOUR-PROJECT-NAME**.

You'll want to edit the content though, right? Add your cat pictures or resume or pizza recipes? You can do that.

You can create, edit, move, rename, and delete files all through the GitHub website. Check out these blog posts on GitHub for details on how to do those things: - Create files[6] - Edit files[7] - Move and rename files[8] - Delete files[9]

You can also clone the project repository onto your computer:

```
1 git clone git@github.com:__YOUR-USERNAME__/__YOUR-PROJECT-NAME__.git
```

You can copy the git url to clone from the right-hand sidebar of your project repository.

After cloning the repository, `cd` into it and make some changes:

[3] https://help.github.com/categories/20/articles
[4] http://pages.github.com
[5] https://github.com/maxogden/gh-pages-template
[6] https://github.com/blog/1327-creating-files-on-github
[7] https://github.com/blog/143-inline-file-editing
[8] https://github.com/blog/1436-moving-and-renaming-files-on-github
[9] https://github.com/blog/1545-deleting-files-on-github

```
cd __YOUR-PROJECT-NAME__
nano index.html
```

Add a bunch of content to index.html, and change the styles in style.css.

After you've made some changes, add them to the repo and commit the changes:

```
git add .
git commit -m 'include a brief, clear message about the changes'
```

Now, push your changes back to GitHub:

```
git push origin gh-pages
```

8 Get started

To get ready for working through the upcoming examples, we'll set up a few things.

Make sure all the software we installed is running

vagrant

Running `vagrant --help` should return the help text of vagrant.

git

Running the command `git --help` should return the help text of git.

sublime

Open up the Sublime text editor to ensure that it is installed properly.

Create a folder for working through examples

Let's make a folder named dev-envs in your home directory.

On Macs the home directory is:

```
/Users/YOUR-USERNAME
```

On Windows:

```
%userprofile%
```

On Linux:

```
/home/YOUR-USERNAME
```

Shortcut

On Mac, Linux, and recent versions of Windows (in the Powershell terminal / in Windows 7+), there's a useful alias for a user's home directory, the tilde:

```
~
```

So, you can run a command like `cd ~`, and that'll take you to your home directory.

Once you've navigated to your home folder, create the dev-envs folder:

```
mkdir dev-envs
```

Change directory into dev-envs, then create directories for javascript, ruby, and python:

```
cd dev-envs
mkdir javascript ruby python
```

We'll use these directories to store the examples we work through later in the book.

9 Ruby

Ruby is a pleasant language that gets about as close to the English language as possible. It's popular in large part because of the web development framework Ruby on Rails.

Language website

http://www.ruby-lang.org/en[1]

Documentation

docs: http://www.ruby-lang.org/en/documentation[2]

Vagrant

Let's create a vagrant machine in your ruby dev-envs folder:

```
mkdir ~/dev-envs/ruby
cd ~/dev-envs/ruby
```

Create a new vagrant machine using the Ubuntu Precise box:

```
vagrant init precise32
```

Now start the vagrant machine:

```
vagrant up
```

If all goes well that'll result in output similar to the following:

[1] http://www.ruby-lang.org/en
[2] http://www.ruby-lang.org/en/documentation

```
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'precise32'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Fixed port collision for 22 => 2222. Now on port 2200.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2200 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

Now we will log in to the vagrant machine. This will be very much like using the `ssh` command to log in to a remote server.

Use this command:

```
vagrant ssh
```

You should see output similar to the following:

```
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:22:31 2012 from 10.0.2.2
```

We'll now install ruby and related tools, and get started building applications.

Complete all the following instructions while logged in to the vagrant machine.

Install git & dependencies

To get started, we'll need to install git and some necessary system dependencies while logged in to the virtual machine:

```
sudo apt-get install git gcc make zlib1g zlib1g-dev
```

Installing ruby

We'll be using rbenv to install ruby: https://github.com/sstephenson/rbenv[3].

We'll also need ruby-build: https://github.com/sstephenson/ruby-build[4].

Install rbenv into ~/.rbenv.

```
git clone https://github.com/sstephenson/rbenv.git ~/.rbenv
```

Make sure ~/.rbenv/bin is in your $PATH so you can use the rbenv command-line utility.

```
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
```

To use shims and autocompletion with rbenv, add rbenv init to your ~/.bashrc file.

```
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
```

Source the ~/.bashrc file so that the rbenv command is available.

```
source ~/.bashrc
```

Check if rbenv was set up by running the rbenv command:

```
rbenv
```

If rbenv was successfully installed, you'll see the following help output:

[3] https://github.com/sstephenson/rbenv
[4] https://github.com/sstephenson/ruby-build

```
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
   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/sstephenson/rbenv#readme
```

Install ruby-build

In order to install different versions of ruby using rbenv, we'll install the ruby-build tool.

```
git clone https://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-bui\
ld
```

Now we can install a new version of ruby using the `ruby install` command.

To see a list of the names of all possible ruby versions and implementations you can install, run this command:

```
rbenv install -l
```

Let's install the latest version of ruby 2.0, which as of this writing is 2.0.0-p247 using this command:

```
rbenv install 2.0.0-p247
```

This will download and install the latest ruby. It'll take a while, so take a break for a few minutes.

If you want to set this new ruby version as the default, which I recommend doing for now, run this command:

```
rbenv global 2.0.0-p247
```

This sets ruby 2.0.0-p247 as the global ruby, so it'll always be the version used with your projects in this vagrant machine.

Install rbenv-gem-rehash

By default you would need to run `rbenv rehash` every time you install new gems to set up rbenv shims for each of the bin commands associated with the new gems. This rbenv plugin makes it so you don't have to run `rbenv rehash` each time.

```
git clone https://github.com/sstephenson/rbenv-gem-rehash.git ~/.rbenv/plugins/rb\
env-gem-rehash
```

Package manager: rubygems

To install ruby packages, you'll use the `gem` command.

For example, basic `gem` command usage looks like this:

```
gem install some-package-name
```

As an example, we'll install the bundler gem, which we'll put to use later:

```
gem install bundler
```

Build tools / automating repetitive tasks

For automating tasks in ruby development, use rake[5].

Install

First, install the rake gem:

```
gem install rake
```

Create a Rakefile for your project:

```
touch Rakefile
```

Add this simple example to your Rakefile:

[5] http://rake.rubyforge.org/

```
task :default => [:start]

task :start do
  ruby "app.rb"
end
```

When you run this command on the terminal:

```
rake
```

The start task defined in your Rakefile will be executed.

Learn more about rake by reading the project documentation[6].

Testing: minitest

https://github.com/seattlerb/minitest

Language basics

variables

Create a variable like this:

```
a = 1
```

numbers

In ruby there are integers and floats. An integer is a whole number, a float is a decimal.

Integers:

```
1
100
223239
```

Floats:

[6] http://rake.rubyforge.org/

```
1.0
5.132
3.14
```

string

A string is a *string of characters* wrapped in single or double quotes:

```
"this is a string"
'this is also a string'
```

When using double quotes, you can use string interpolation to insert the values of variables into a string:

```
food = 'pizza'
sentence = "#{food} is yummy."
```

The `sentence` variable will return this: `pizza is yummy`.

array

An array is like a list of values. You can put anything in an array: strings, numbers, other arrays, hashes, etc.

They look like this:

```
things = ['pizza is great.', 30, ['yep', 'ok']]
```

Accessing values in an array:

You can access values in an array by typing the variable name followed by square brackets and a number that represents the value you'd like to access.

Arrays are zero-indexed, so the first item is represented by a zero:

```
things[0]
```

This returns `'pizza is great'`.

To get at nested arrays, you add another set of square brackets, like this:

```
things[2][1]
```

The above statement returns `'ok'`.

hash

A hash is much like an array, except instead of the values being indexed by numbers, they have names.

A simple hash looks like this:

```
pizza = { :tastes => 'really good', :slices_i_can_eat => 100 }
```

There's also an alternate, more concise syntax for creating a hash that looks like this:

```
pizza = { tastes: 'really good', slices_i_can_eat: 100 }
```

Accessing hash values

Accessing values in a hash looks similar to arrays:

```
pizza[:tastes]
```

The above statement returns `'really good'`.

```
pizza[:slices_i_can_eat]
```

The above statement returns `100`. That's a lot of slices of pizza.

function

A function definition is super simple:

```
def eat(food)
  return "I ate #{food}."
end
```

`def` indicates that we're about to define a function.

`eat` is the name of the function.

In parentheses, we indicate that one argument is expected, `food`.

This line: `return "I ate #{food}."` indicates that `"I ate #{food}."` will be returned when the function is called.

We can actually rewrite that line to exclude the `return` keyword. The last statement in a function will get returned automatically. Since we only have one line in this function we don't need 'return'.

Calling a function

Now that the function is defined, you can call it like this:

```
eat('pizza')
```

That statement will return: `'I ate pizza.'`.

The parenthese are optional, so we can write the function call like this:

```
eat 'pizza'
```

class

Define a class in ruby like this:

```
class Meal
  # here we'll define methods on the class
end
```

class methods

Class methods are basically functions that exist inside of a class namespace.

```
class Meal

  def initialize(food)
    @food = food
  end

  def prepare
    @prepared = true
    "#{@food} is ready!"
  end

  def eat
    if @prepared == true
      "dang, that #{@food} sure was good."
    else
      "relax, the #{@food} isn't prepared yet."
    end
  end

end
```

class instance

Now, to use our Meal class and call the `prepare` and `eat` methods, we do this:

```
1 dinner = Meal.new 'pizza'
```

We can call a method by typing the name of the class instance, followed by a period and the name of the method. Let's try out the `eat` method:

```
1 dinner.eat
```

That's return this string: `"relax, the pizza isn't prepared yet."`.

So let's `prepare` the dinner:

```
1 dinner.prepare
```

That'll return this string: `"pizza is ready!"`.

Now run `dinner.eat` and we'll see this string: `"dang, that pizza sure was good."`.

importing/requiring code

We can require the functionality of ruby gems and code from other files by using the `require` statement, typically at the top of the file. An example of requiring the sinatra gem:

```
1 require 'sinatra'
```

Web framework: sinatra

Project website: http://www.sinatrarb.com[7].

Install

Navigate to your ruby projects folder:

```
1 cd ~/dev-envs/ruby
```

Create a folder named hello-sinatra and navigate into it:

[7] http://www.sinatrarb.com/

```
mkdir hello-sinatra && cd hello-sinatra
```

```
gem install sinatra
```

Simple example

Inside your hello-sinatra directory, create a file named app.rb:

```
touch app.rb
```

Here's a simple example of a sinatra app:

```
require 'sinatra'

get '/' do
  'pizza is awesome'
end
```

Type that code into your app.rb file.

Now you can run your app with this command:

```
ruby app.rb
```

Let's go through this app line by line:

Import the sinatra functionality into our app:

```
require 'sinatra'
```

Call `get`, to respond to requests for the root url:

```
get '/' do
```

Respond to requests with some text:

```
  'pizza is awesome'
```

Close the `get` block:

```
end
```

Extended example

Let's make a small website with sinatra[8] to explore how it works.

In this example our site will do three things:
- serve html at the root route from a view that has a list of posts - serve html for a single post at `/post/:id` - serve json at `/api/posts` that has a list of posts

We won't be using a database for this example, but instead will use a json file with a list of posts.

To get started, create and change directory into a new project folder.

```
mkdir sinatra-example
cd sinatra-example
```

We'll be using sinatra and will utilize the default template language, erb. Let's install sinatra by creating a Gemfile:

```
touch Gemfile
```

Add the sinatra gem to the Gemfile:

```
gem 'sinatra'
```

Now run bundle to install sinatra:

```
bundle
```

We will use shotgun[9] to run the app – shotgun will automatically restart the server each time you edit a file in the project.

Install shotgun:

```
gem install shotgun
```

To run the sinatra app you'll use this command:

[8] http://sinatrarb.com
[9] https://github.com/rtomayko/shotgun

```
shotgun app.rb
```

Create a file named posts.json with the following json:

```
[
{
  "title": "This is the first post",
  "slug": "first-post",
  "content": "The pizza is awesome. The pizza is awesome. The pizza is awesome. T\
he pizza is awesome. The pizza is awesome. The pizza is awesome. The pizza is awe\
some. The pizza is awesome. The pizza is awesome. The pizza is awesome. The pizza\
 is awesome."
},
{
  "title": "Another post that you might like",
  "slug": "second-post",
  "content": "Eating pizza is great. Eating pizza is great. Eating pizza is great\
. Eating pizza is great. Eating pizza is great. Eating pizza is great. Eating piz\
za is great. Eating pizza is great. Eating pizza is great. Eating pizza is great.\
 Eating pizza is great. Eating pizza is great."
},
{
  "title": "The third and last post",
  "slug": "third-post",
  "content": "The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out."
}
]
```

Now create the app.rb file:

```
require 'sinatra'
require 'json'

before do
  @title = 'Extended Sinatra example'
  @posts = JSON.parse( IO.read('posts.json') )
end

get '/' do
  erb :index
end

get '/post/:slug' do
  @posts.each do |post|
    if post['slug'] == params[:slug]
      @post = post
    end
  end
  erb :post
end

get '/api/posts' do
  data = {
    meta: { name: @title },
    posts: @posts
  }
  data.to_json
end
```

Let's break down this example code chunk by chunk:

Require the necessary ruby libraries:

```
require 'sinatra'
require 'json'
```

Create global variables that are available to our views using the before method, which runs before a request is processed:

```
before do
  @title = 'Extended Sinatra example'
  @posts = JSON.parse( IO.read('posts.json') )
end
```

Serve the index.erb view on the root url with the following code block. note that an erb view is rendered using the `erb` method, and you don't have to include the .erb file suffix. Sinatra automatically looks in a folder named views, so you only have to pass the file name:

```
get '/' do
  erb :index
end
```

The following code block listens for requests for a specific blog post. We iterate through each of the items in our posts array, and if the slug that's passed in the url matches a slug in the posts array, that post is set to a global `@post` variable that's available in our post view.

```
get '/post/:slug' do
  @posts.each do |post|
    if post['slug'] == params[:slug]
      @post = post
    end
  end
  erb :post
end
```

The following is a simple example of exposing a simple json feed of the posts:

```
get '/api/posts' do
  data = {
    meta: { name: @title },
    posts: @posts
  }
  data.to_json
end
```

Next, we'll need the erb views for rendering html.

Let's make a views folder for all the views to live in:

```
mkdir views
```

And create all the view files that we need:

```
touch views/layout.erb views/index.erb views/post.erb
```

Add this content to the layout.erb file:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title><%= @title %></title>
  <link rel="stylesheet" href="/styles.css">
</head>
<body>

<header>
  <div class="container">
    <h1><a href="/"><%= @title %></a></h1>
  </div>
</header>

<main role="main">
  <div class="container">
    <%= yield %>
  </main>
</div>

<footer>
  <div class="container">
    <p>Posts are also available via json at <a href="/api/posts">/api/posts</a>/
  </div>
</footer>

</body>
</html>
```

Add this content to the index.erb file:

```
<% for @post in @posts %>
  <h3>
    <a href="/post/<%= @post['slug'] %>">
      <%= @post['title'] %>
    </a>
  </h3>
  <div><%= @post['content'] %></div>
<% end %>
```

Add this content to the post.erb file:

```
<h3><%= @post['title'] %></h3>
<div><%= @post['content'] %></div>
```

Let's add some css styling so this looks a little more readable.

First create the public folder and the styles.css file:

```
mkdir public
touch public/styles.css
```

Now add this content to the styles.css file:

```
body {
  font: 16px/1.5 'Helvetica Neue', Helvetica, Arial, sans-serif;
  color: #787876;
}

h1, h3 {
  font-weight: 300;
  margin-bottom: 5px;
}

a {
  text-decoration: none;
  color: #EA6045;
}

a:hover {
  color: #2F3440;
}

```

```
.container {
  width: 90%;
  margin: 0px auto;
}

footer {
  margin-top: 30px;
  border-top: 1px solid #efefef;
  padding-top: 20px;
  font-style: italic;
}

@media (min-width: 600px){
  .container {
    width: 60%;
  }
}
```

You should now be able to navigate the home page, three blog post pages, and the posts json feed. Run the project with the nodemon command:

```
shotgun app.rb
```

Sinatra resources

Learn more about sinatra at the sinatra website: http://www.sinatrarb.com[10]

Resources

http://tryruby.org/

[10] http://www.sinatrarb.com/

10 Javascript

Javascript is a ubiquitous language. It's getting first class support in various operating sytems, you can use it on the server with node.js, and developing javascript applications that are largely front-end code is becoming a popular and pragmatic practice.

We'll cover javascript on the server using node.js, and briefly cover javascript in the browser.

Javascript development tools have recently seen a big burst of growth thanks to node.js.

Language website

The website for node.js is pretty great: http://nodejs.org[1].

It includes the API documentation, a blog that announces new releases and a link to the website for the package manager used by node developers, npm.

There isn't really a website for the javascript language in the same way ruby and python have websites. For general javascript information look to the documentation websites listed below:

Documentation

One of the best places to start learning Node.js is nodeschool.io[2]. These are a set of interactive workshops you complete using the terminal. Highly recommended.

The node.js API documentation: http://nodejs.org/api/index.html[3]

There are a lot of different resources for client-side javascript documentation.

I recommend two:

Web Platform Docs: http://www.webplatform.org/[4]

The Web Platform Docs is a relatively new set of documentation that includes coverage of html, css, and javscript. It's pretty good,

Mozilla Developer Network documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript[5]

The MDN docs are great. There are bits that are specific to Mozilla, but the majority of the content is relevant to html, css, and javascript in general.

---

[1] http://nodejs.org

[2] http://nodeschool.io/

[3] http://nodejs.org/api/index.html

[4] http://www.webplatform.org/

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript

Vagrant

Let's create a vagrant machine in your javascript dev-envs folder:

```
mkdir ~/dev-envs/javascript
cd ~/dev-envs/javascript
```

Create a new vagrant machine using the Ubuntu Precise box:

```
vagrant init precise32
```

Now start the vagrant machine:

```
vagrant up
```

If all goes well that'll result in output similar to the following:

```
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'precise32'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Fixed port collision for 22 => 2222. Now on port 2200.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2200 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

Now we will log in to the vagrant machine. This will be very much like using the `ssh` command to log in to a remote server.

Use this command:

```
vagrant ssh
```

You should see output similar to the following:

```
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:22:31 2012 from 10.0.2.2
```

We'll now install Node.js, its dependencies, and related tools, and get started building applications. Complete all the following instructions while logged in to the vagrant machine.

Install git & dependencies

To get started, we'll need to install git and some necessary system dependencies while logged in to the virtual machine:

```
sudo apt-get install git gcc make curl
```

Installing node.js

I recommend using a tool called `nvm` for installing node.js if you're on mac or linux. It's very similar to the rbenv tool we used in the last chapter for installing ruby.

If you're on Windows, install node.js using the .msi package on the nodejs.org downloads page: http://nodejs.org/downloads[6].

nvm: https://github.com/creationix/nvm[7].

We have git installed, so we can clone nvm to our home folder:

```
git clone https://github.com/creationix/nvm.git ~/.nvm
```

Source nvm to make it the `nvm` command available in the terminal:

```
source ~/.nvm/nvm.sh
```

To ensure that `nvm` is available at all times in the terminal, add the above line to your ~/.bashrc file:

[6] http://nodejs.org/downloads
[7] https://github.com/creationix/nvm

```
nano ~/.bashrc
```

Add `source ~/.nvm/nvm.sh` to the ~/.bashrc file.

To get the `nvm` command after adding that line to your ~/.bashrc file, source your ~/.bashrc file:

```
source ~/.bashrc
```

To ensure `nvm` is working, run the command without options:

```
nvm
```

You should see output like this:

```
Node Version Manager

Usage:
    nvm help                    Show this message
    nvm install [-s] <version>  Download and install a <version>
    nvm uninstall <version>     Uninstall a version
    nvm use <version>           Modify PATH to use <version>
    nvm run <version> [<args>]  Run <version> with <args> as arguments
    nvm ls                      List installed versions
    nvm ls <version>            List versions matching a given description
    nvm ls-remote               List remote versions available for install
    nvm deactivate              Undo effects of NVM on current shell
    nvm alias [<pattern>]       Show all aliases beginning with <pattern>
    nvm alias <name> <version>  Set an alias named <name> pointing to <version>
    nvm unalias <name>          Deletes the alias named <name>
    nvm copy-packages <version> Install global NPM packages contained in <version\
> to current version

Example:
    nvm install v0.4.12         Install a specific version number
    nvm use 0.2                 Use the latest available 0.2.x release
    nvm run 0.4.12 myApp.js     Run myApp.js using node v0.4.12
    nvm alias default 0.4       Auto use the latest installed v0.4.x version
```

The above help text gives a good overview of usage of the `nvm` command.

Now we install node.js

Install the latest version of node v0.10.x:

```
nvm install 0.10
```

You'll see output like this:

```
######################################################################## 100.0%
Now using node v0.10.21
```

We can switch to that new version using this command:

```
nvm use 0.10.21
```

And to set that version as the default, set the default alias:

```
nvm alias default 0.10.21
```

Javascript in the browser

You don't need to install anything for javascript in the browser. The browser takes care of that for you. I recommend using Chrome for the examples in this book. Firefox is also excellent, and if you choose to use it, there will be just slight differences between the developer tools compared to Chrome.

Download Chrome here: https://www.google.com/intl/en/chrome/browser/[8]

Package manager: npm

When you install node.js, you get npm.

npm: http://npmjs.org[9]

You may also want to use bower[10] or component[11], two package managers that specifically target client-side code. Remember that javascript packages distributed via npm are not limited to node.js, and can also be used in the browser in many cases through the use of module bundlers like browserify[12] and webpack[13].

[8] https://www.google.com/intl/en/chrome/browser/
[9] http://npmjs.org
[10] http://bower.io/
[11] http://component.io
[12] http://browserify.org
[13] http://webpack.github.io/

Build tools / automating repetitive tasks

There are a few ways to automate repetitive tasks in JavaScript projects.

npm scripts

Using npm scripts and the `npm run` command is a clean, simple method for organizing the build tools of your JavaScript project.

You specify npm scripts by adding to the `scripts` field of a package.json file in your JavaScript project.

Take this example:

```
"scripts": {
  "test": "node test.js",
  "start": "node server.js",
  "bundle": "browserify main.js -o bundle.js"
}
```

We would run `npm test` to test the code, `npm start` to run a development server, and `npm run bundle` to create a bundled JavaScript file using the `browserify` command.

Grunt

Another, more complicated option is grunt.js[14].

Install

First, install the grunt command-line tool:

```
npm install -g grunt-cli
```

Next, you'll create a Gruntfile.js in your project.

Learn more about grunt.js by reading the project documentation[15].

[14] http://gruntjs.com

[15] http://gruntjs.com/getting-started

More information about npm scripts and Grunt

Check out this blog post for more information about npm scripts, Grunt, and how I choose between the two: http://superbigtree.tumblr.com/post/59519017137/introduction-to-grunt-js-and-npm-scripts-and-choosing[16]

Testing: tape

For testing, we'll use a library named tape.

tape: https://github.com/substack/tape[17].

Installing tape

To install tape, we'll use `npm` on the command line.

Open a terminal.

Change directory to your projects folder.

```
cd ~/Projects
```

Create a directory for our first javascript project:

```
mkdir learn-javascript-one
```

Change directory into our new project folder:

```
cd learn-javascript-one
```

Run this command to create a package.json file:

```
npm init
```

Answer the questions that pop up.

Now, to really install tape:

[16] http://superbigtree.tumblr.com/post/59519017137/introduction-to-grunt-js-and-npm-scripts-and-choosing
[17] https://github.com/substack/tape

```
npm install --save-dev tape
```

`npm install` is used to install packages from npm.

`--save-dev` saves the package to your package.json as a development dependency.

`tape` is the package name. You can pass multiple packages at once, separated by commas.

A simple example of a test written with tape:

```
var test = require('tape');

var p = 'pizza';

test('pizza test', function (t) {
  t.plan(1);

  t.equal(p, 'pizza');
});
```

Let's go through this line-by-line in a high-level way.

Here we assign the tape functionality to a variable named `test`:

```
var test = require('tape');
```

Here `p` is assigned to the string `'pizza'`:

```
var p = 'pizza';
```

Now we're calling `test`, and describing it as a `pizza test`:

```
test('pizza test', function (t) {
```

We're given the argument `t` to use to call testing methods.

Here we specify that we plan to have 1 test in our code:

```
  t.plan(1);
```

Here's that one test, making sure that the `p` variable is equal to the string `pizza`:

```
  t.equal(p, 'pizza');
```

This closes the function:

```
});
```

Those are the very basics of using tape. Next, we'll dive deeper into some javascript basics, and use tape to test our code.

Language basics

Variables

Creating a variable:

```
var nameOfVariable;
```

> Variables are camelCase, meaning first letter is lowercase, and if the variable is made of multiple words, the first letter of following words are capitalized.

Creating a variable that references a string:

```
var thisIsAString = 'this is a string';
```

Surround strings with single quotes.

Creating a variable that references a number:

```
var thisIsANumber = 3.14;
```

Numbers do not have quotes around them.

Creating a variable that references an array:

```
var thisIsAnArray = [1, "two", [3, 4]];
```

Note that one of the values in the array is a number, one is a string, and another is an array. Arrays can hold any value in any order.

Accessing the values in an array:

```
thisIsAnArray[0];
```

The above will return the number `1`. Arrays use numbers as the index of their values, and with javascript an array's index always start at `0`, making `0` reference the first value of the array.

```
thisIsAnArray[1];
```

This returns the string 'two';

How would you return the number `4` from the nested array? Like this:

```
thisIsAnArray[2][1];
```

Creating a variable that references an object:

var thisIsAnObject = { someString: 'some string value', someNumber: 1234, someFunction: function(){ return 'a function that belongs to an object'; } }

Here we're setting `someString` to `'some string value',someNumber' to 1234, and we're creating a function named` someFunction `that returns the string` 'a function that belongs to an object''. So how do we access these values?

To get the value of `someString` using dot notation:

```
thisIsAnObject.someString;
```

Or using bracket notation:

```
thisIsAnObject['someString'];
```

To get the value of `someNumber` using dot notation:

```
thisIsAnObject.someNumber;
```

Or using bracket notation:

```
thisIsAnObject['someNumber'];
```

To use the function `someFunction` using dot notation:

```
thisIsAnObject.someFunction();
```

Or using bracket notation:

```
thisIsAnObject['someFunction']();
```

Using square bracket notations with functions looks a little wacky. It will be useful if you are storing function names in variables as strings, and need to use the variable to call the function being stored. Otherwise, stick with dot notation. That goes for other attributes on an object, too: stick with dot notation unless there's a good reason to use bracket notation.

For instance, it's more clear to use bracket notation in a situation like this:

```
for (var key in object){
  thisIsAnObject[key];
}
```

This gives you an idea of how to iterate through an object using a for...in loop.

importing/requiring code

Node.js

When using Node.js we can require the functionality of Node.js modules distributed via npm and code from other files by using the `require` function, typically at the top of the file. An example of requiring the express module:

```
var express = require('express');
```

Browser

For browser side code we might add a script tag into the HTML file of our project. Here's an example of a script tag:

```
<script src="main.js"></script>
```

Alternately we might use a tool like browserify to require packages using the same method as Node.js. Learn more about browserify at the project website, browserify.org[18].

[18] http://browserify.org

Web framework: express

Express is a small web framework for node.js, originally inspired by sinatra.

express: http://expressjs.com/[19].

Install

Navigate to your javascript projects folder:

```
cd ~/dev-envs/javascript
```

Create a folder named hello-express and navigate into it:

```
mkdir hello-express && cd hello-express
```

```
npm install express
```

This installs express locally so you can use it in your app.

Simple example

Inside your hello-express directory, create a file named app.js:

```
touch app.js
```

Here's a simple example of an express app:

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  res.send('pizza is awesome.');
});

app.listen(3000);

console.log('app is listening at localhost:3000');
```

Type that code into your app.js file

You can run your app with this command:

[19] http://expressjs.com/

```
node app.js
```

Now let's run through app.js one line at a time:

Save the express module to a variable named express:

```
var express = require('express');
```

Create our app by calling `express()` and assigning the returned object to the variable `app`:

```
var app = express();
```

Exposing a route for the rool url using `app.get()`:

```
app.get('/', function(req, res){
```

`req` is an argument we get from the callback that represents the request. `res` represents the response that we'll be sending back to the user.

Sending a text response:

```
  res.send('pizza is awesome.');
```

Closing the `app.get()` function:

```
});
```

Setting up the app to listen for requests on port 3000:

```
app.listen(3000);
```

Logging a message to the user on the console so that the user knows that the app has started and things are happening:

```
console.log('app is listening at localhost:3000');
```

Installing express globally

```
npm install -g express
```

The `-g` option installs express globally. This means that there is now an `express` command available in your terminal you can use to create a new express app.

Go ahead and install express globally using the above command, and we'll continue experimenting with express.

Create an app using the `express` command.

Navigate to your Projects folder and run this command:

```
express new-app
```

This will generate a bunch of files for you. I won't go into the details of what's created, but it's good to know express has this functionality available.

Extended example

Let's make a small website with express[20] to explore how it works.

In this example our site will do three things:
- serve html at the root route from a view that has a list of posts - serve html for a single post at `/post/:id` - serve json at `/api/posts` that has a list of posts

We won't be using a database for this example, but instead will use a json file with a list of posts.

To get started, create and change directory into a new project folder, then run `npm init` to create a package.json file.

```
mkdir express-example
cd express-example
```

We'll be using express and for templates we'll use the ejs[21] module, so let's install those dependencies:

```
npm install --save express ejs
```

We will use nodemon[22] to run the app – nodemon will automatically restart the server each time you edit a file in the project.

Install nodemon:

[20] http://expressjs.com
[21] https://github.com/visionmedia/ejs
[22] https://github.com/remy/nodemon

```
npm install -g nodemon
```

Run nodemon with these options so that changes to ejs views and public files also trigger the restart:

```
nodemon -e js,css,html,ejs
```

Create a file named posts.json with the following json:

```
[
{
  "title": "This is the first post",
  "slug": "first-post",
  "content": "The pizza is awesome. The pizza is awesome. The pizza is awesome. T\
he pizza is awesome. The pizza is awesome. The pizza is awesome. The pizza is awe\
some. The pizza is awesome. The pizza is awesome. The pizza is awesome. The pizza\
 is awesome."
},
{
  "title": "Another post that you might like",
  "slug": "second-post",
  "content": "Eating pizza is great. Eating pizza is great. Eating pizza is great\
. Eating pizza is great. Eating pizza is great. Eating pizza is great. Eating piz\
za is great. Eating pizza is great. Eating pizza is great. Eating pizza is great.\
 Eating pizza is great. Eating pizza is great."
},
{
  "title": "The third and last post",
  "slug": "third-post",
  "content": "The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out."
}
]
```

First we'll create the app.js file:

```
var express = require('express');
var fs = require('fs');
var app = express();

app.use('/public', express.static(__dirname + '/public'));

app.locals({
  title: 'Extended Express Example'
});

app.all('*', function(req, res, next){
  fs.readFile('posts.json', function(err, data){
    app.locals.posts = JSON.parse(data);
    next();
  });
});

app.get('/', function(req, res){
  res.render('index.ejs');
});

app.get('/post/:slug', function(req, res, next){
  app.locals.posts.forEach(function(post){
    if (req.params.slug === post.slug){
      res.render('post.ejs', { post: post });
    }
  })
});

app.get('/api/posts', function(req, res){
  var data = {
    meta: { name: app.locals.title },
    posts: app.locals.posts
  }
  res.json(data);
});

app.listen(3000);
console.log('app is listening at localhost:3000');
```

Let's break down this example code chunk by chunk:

Require the needed modules and create the app variable:

```
var express = require('express');
var fs = require('fs');
var app = express();
```

Set up the app to serve whatever is in the public folder at the url `/public/:filename`:

```
app.use('/public', express.static(__dirname + '/public'));
```

You can add local variables that can be used in views and throughout the app by passing an object to `app.locals()`:

```
app.locals({
  title: 'Extended Express Example'
});
```

In this example we're loading the posts from the json file before responding to routes:

```
app.all('*', function(req, res, next){
  fs.readFile('posts.json', function(err, data){
    app.locals.posts = JSON.parse(data);
    next();
  });
});
```

When a browser requests the root url, our app responds with the index.ejs file. Express automatically looks in a folder named views, so you only have to pass the file name:

```
app.get('/', function(req, res){
  res.render('index.ejs');
});
```

The following code block listens for requests for a specific blog post. We iterate through each of the items in our posts array, and if the slug that's passed in the url matches a slug in the posts array, that post is returned:

```
app.get('/post/:slug', function(req, res, next){
  app.locals.posts.forEach(function(post){
    if (req.params.slug === post.slug){
      res.render('post.ejs', { post: post });
    }
  })
});
```

The following is a simple example of exposing a simple json feed of the posts.

```
app.get('/api/posts', function(req, res){
  var data = {
    meta: { name: app.locals.title },
    posts: app.locals.posts
  }
  res.json(data);
});
```

And finally, we make the app listen on port 3000, and print a message to the terminal:

```
app.listen(3000);
console.log('app is listening at localhost:3000');
```

Next, we'll need the views for rendering html. We'll use a templating language named ejs for our views.

The only downside to ejs is that it doesn't allow us to specify a layout view like we did with sinatra and erb.

To get around that we'll create header and footer views that we later include on other views.

Let's make a views folder for all the views to live in:

```
mkdir views
```

And create all the view files that we need:

```
touch views/header.ejs views/footer.ejs views/index.ejs views/post.ejs
```

Add this content to the header.ejs file:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title><%= title %></title>
  <link rel="stylesheet" href="/public/styles.css">
</head>
<body>

<header>
  <div class="container">
    <h1><a href="/"><%= title %></a></h1>
  </div>
</header>
```

Add this content to the footer.ejs file:

```
<footer>
  <div class="container">
    <p>Posts are also available via json at <a href="/api/posts">/api/posts</a>/
  </div>
</footer>

</body>
</html>
```

Add this content to the index.ejs file:

```
<% include header %>

<main role="main">
  <div class="container">
    <% posts.forEach(function(post){ %>
      <h3>
        <a href="/post/<%= post.slug %>">
          <%= post.title %>
        </a>
      </h3>
      <div><%= post.content %></div>
    <% }); %>
  </main>
```

```
</div>

<% include footer %>
```

Add this content to the post.ejs file:

```
<% include header %>

<main role="main">
  <div class="container">
    <h3><%= post.title %></h3>
    <div><%= post.content %></div>
  </main>
</div>

<% include footer %>
```

Let's add some css styling so this looks a little more readable.

First create the public folder and the styles.css file:

```
mkdir public
touch public/styles.css
```

Now add this content to the styles.css file:

```
body {
  font: 16px/1.5 'Helvetica Neue', Helvetica, Arial, sans-serif;
  color: #787876;
}

h1, h3 {
  font-weight: 300;
  margin-bottom: 5px;
}

a {
  text-decoration: none;
  color: #EA6045;
}

a:hover {
```

```
  color: #2F3440;
}

.container {
  width: 90%;
  margin: 0px auto;
}

footer {
  margin-top: 30px;
  border-top: 1px solid #efefef;
  padding-top: 20px;
  font-style: italic;
}

@media (min-width: 600px){
  .container {
    width: 60%;
  }
}
```

You should now be able to navigate on the home page, three blog post pages, and the posts json feed. Run the project with the nodemon command:

```
nodemon -e js,css,html,ejs
```

Express resources

Learn more about express at the express website: http://expressjs.com[23]

Try out the ExpressWorks workshop on nodeschool.io: nodeschool.io/#expressworks[24]

[23] http://expressjs.com/

[24] http://nodeschool.io/#expressworks

11 Python

Python is a language that is readable, quick to learn, and used for a wide range of purposes, including web development, science, and in academia.

In this chapter we'll review some basics of the Python language, testing with UnitTest, creating dev environments with pip and virtualenv, and building apps with flask, a small web development framework.

We'll be working with Python version 2.7, which should be installed by default on most systems, and is still best supported by various Python libraries. In the future we'll do an update to this book to support Python 3.

Language website

http://www.python.org/

Documentation

http://www.python.org/doc/

Vagrant

Let's create a vagrant machine in your python dev-envs folder:

```
1 mkdir ~/dev-envs/python
2 cd ~/dev-envs/python
```

Create a new vagrant machine using the Ubuntu Precise box:

```
1 vagrant init precise32
```

Now start the vagrant machine:

```
vagrant up
```

If all goes well that'll result in output similar to the following:

```
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'precise32'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Fixed port collision for 22 => 2222. Now on port 2200.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2200 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

Now we will log in to the vagrant machine. This will be very much like using the `ssh` command to log in to a remote server.

Use this command:

```
vagrant ssh
```

You should see output similar to the following:

```
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:22:31 2012 from 10.0.2.2
```

We'll now install python, its dependencies, and related tools, and get started building applications. Complete all the following instructions while logged in to the vagrant machine.

Install git & dependencies

To get started, we'll need to install git and some necessary system dependencies while logged in to the virtual machine:

```
sudo apt-get install git python-setuptools python-dev build-essential
```

Installing python

Python is most likely already installed on your machine.

Package manager: pip

pip: http://www.pip-installer.org/en/latest/[1] https://github.com/pypa/pip

Use virtualenv

virtualenv: http://www.virtualenv.org/en/latest/[2]

Build tools / automating repetitive tasks

For automating tasks in python development, use fabric[3].

Install

First, install fabric:

```
pip install fabric
```

Create a fabfile.py in your project directory:

```
touch fabfile.py
```

Add this example to your fabfile.py:

[1] http://www.pip-installer.org/en/latest/
[2] http://www.virtualenv.org/en/latest/
[3] http://fabfile.org

```
from fabric.api import local

def start():
    local("python app.py")
```

Run this command:

```
fab start
```

The start task defined in your fabfile.py will be executed.

Learn more about fabric by reading the project documentation[4].

Testing: unittest

We'll be using unittest as the testing framework with python. It comes bundled with python so it doesn't have to be installed separately.

unittest documentation: http://docs.python.org/2.7/library/unittest.html[5]

Here's a very simple example of unittest usage:

A simple example of a test written with tape:

```
import unittest

class PizzaTest(unittest.TestCase):

    def setUp(self):
        self.pizza = 'pizza'

    def test_pizza(self):
        self.assertEqual(self.pizza, 'pizza')

if __name__ -- '__main__':
    unittest.main()
```

Language basics

variables

Create a variable like this:

[4] http://docs.fabfile.org/en/1.7/

[5] http://docs.python.org/2.7/library/unittest.html

```
some_variable = 'some value'
```

numbers

```
some_number = 3
```

A number is any digit, including decimals, or floating point numbers.

string

```
some_string = 'this is a string'
```

You can create multi-line strings like this:

```
some_big_string = """
This is one line of the string.
And this is another.
This line is also part of the string.
"""
```

A string is text surrounded by single or double quotes.

list

A list in python is very similar to an array in javascript and ruby. Create a list like this:

```
some_list = ['a', 1, 'b']
```

It's possible to nest lists like this:

```
some_nested_list = [1, ['a', 'b', 'c'], 'pizza'];
```

dictionary

Dictionaries in python are similar to objects in javascript or hashes in ruby.

Create a dictionary like this:

```
some_dictionary = { 'thing': 'one', 'otherthing': 'two' }
```

function

Define a function in python like this:

```
def eat(food):
    return 'I ate ' + food
```

class

Create a class in python like this:

```
class Meal:
    # here we define methods on the class
```

method

Defining methods in python looks like this:

```
class Meal:
  def __init__(self, food):
      self.food = food

  def eat(self)
      return 'I ate ' + self.food
```

class instance

Create an instance of the class like this:

```
dinner = Meal('pizza')
dinner.eat()
```

importing/requiring code

To import code into your program, use this syntax:

```
import PACKAGENAME
```

You can import specific classes with this syntax:

```
from PACKAGENAME import CLASS
```

For instance, with the flask library we use later, importing flask looks like this:

```
from flask import Flask
```

Web framework: flask

flask http://flask.pocoo.org/[6]

Navigate to your python projects folder:

```
cd ~/dev-envs/python
```

Create a folder named hello-flask and navigate into it:

```
mkdir hello-flask && cd hello-flask
```

```
pip install flask
```

Simple example

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def pizza():
    return 'pizza is awesome'

if __name__ == '__main__':
    app.run()
```

Type that code into your app.py file.

Now you can run your app with this command:

[6] http://flask.pocoo.org/

```
python app.py
```

Let's look at this little app line by line:

Import the flask functionality into our app:

```
from flask import Flask
```

Create the app by creating an instance of the Flask class:

```
app = Flask(__name__)
```

Define a route for the root url:

```
@app.route('/')
```

Define a function that responds to requests at the

```
def pizza():
    return 'pizza is awesome'
```

The function that immediately follows the `route()` call defines what we'll return when someone visits the root url.

Run the app:

```
if __name__ == "__main__":
    app.run()
```

`app.run()` kicks off a server to serve our app. The `if` statement checks if this code is being executed by the Python interpreter or being included as a module, and app.run() is only called if the code is being executed by the Python interpreter.

Extended example

Let's make a small website with flask[7] to explore how it works.

In this example our site will do three things:
- serve html at the root route from a view that has a list of posts - serve html for a single post at `/post/:id` - serve json at `/api/posts` that has a list of posts

We won't be using a database for this example, but instead will use a json file with a list of posts.

We'll be using the default template language that flask uses, jinja[8]. Let's install flask by creating a virtualenv and using pip:

[7] http://flask.pocoo.org/
[8] http://jinja.pocoo.org/

```
virtualenv flask-example
cd flask-example
```

Activate the virtualenv:

```
source bin/activate
```

Now use pip to install flask and its dependencies

```
pip install flask
```

For this project we'll put the source files inside the virtualenv folder. Create a new folder called source for our application code:

```
mkdir source
cd source
```

Your directory structure should now look like this:

```
flask-example
- bin
- source
- include
- lib
```

Make sure you put new files inside the source folder.

We'll run the application with this command:

```
python app.py
```

Create a file named posts.json with the following json:

```
[
{
  "title": "This is the first post",
  "slug": "first-post",
  "content": "The pizza is awesome. The pizza is awesome. The pizza is awesome. T\
he pizza is awesome. The pizza is awesome. The pizza is awesome. The pizza is awe\
some. The pizza is awesome. The pizza is awesome. The pizza is awesome. The pizza\
 is awesome."
},
{
  "title": "Another post that you might like",
  "slug": "second-post",
  "content": "Eating pizza is great. Eating pizza is great. Eating pizza is great\
. Eating pizza is great. Eating pizza is great. Eating pizza is great. Eating piz\
za is great. Eating pizza is great. Eating pizza is great. Eating pizza is great.\
 Eating pizza is great. Eating pizza is great."
},
{
  "title": "The third and last post",
  "slug": "third-post",
  "content": "The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out. The pizza alw\
ays runs out. The pizza always runs out. The pizza always runs out."
}
]
```

Now create the app.py file:

```
from flask import Flask, render_template, g, jsonify
import json


app = Flask('extended-flask-example')
app.config['DEBUG'] = True


@app.before_request
def before_request():
    g.title = 'Extended flask example'
    posts = open('posts.json')
```

```
    g.posts = json.load(posts)
    posts.close()


@app.route('/')
def index():
    posts = getattr(g, 'posts', None)
    return render_template('index.html', posts=posts)


@app.route('/post/<slug>')
def show_post(slug):
    for post in g.posts:
        if slug == post['slug']:
            return render_template('post.html', post=post)


@app.route('/api/posts')
def show_json():
    meta = { 'name': g.title }
    return jsonify(posts=g.posts, meta=meta)


if __name__ == '__main__':
    app.run()
```

Let's break down this example code chunk by chunk:

Require the necessary python libraries. Note that there are multiple classes from the flask library that we're importing individually:

```
from flask import Flask, render_template, g, jsonify
import json
```

Create the application with a name and turn on debug so the application will reload each time you make changes to the app.py file and provide useful error messages:

```
app = Flask('extended-flask-example')
app.config['DEBUG'] = True
```

Create global variables that are available to our views using the before method, which runs before a request is processed. Here we're loading the posts.json file into the application so we can use it in our views:

```
@app.before_request
def before_request():
    g.title = 'Extended flask example'
    posts = open('posts.json')
    g.posts = json.load(posts)
    posts.close()
```

Serve the index.html template on the root url with the following code block. Note that with flask, templates are the equivalent to views in sinatra or express. Flask automatically looks in a folder named templates, so you only have to specify the filename:

```
@app.route('/')
def index():
    posts = getattr(g, 'posts', None)
    return render_template('index.html', posts=posts)
```

The following code block listens for requests for a specific blog post. We iterate through each of the items in our posts array, and if the slug that's passed in the url matches a slug in the posts array, we render the page with that post set as the `post` variable.

```
@app.route('/post/<slug>')
def show_post(slug):
    for post in g.posts:
        if slug == post['slug']:
            return render_template('post.html', post=post)
```

The following is a simple example of exposing a simple json feed of the posts:

```
@app.route('/api/posts')
def show_json():
    meta = { 'name': g.title }
    return jsonify(posts=g.posts, meta=meta)
```

Finally, this code block checks to make sure our application is not a module being loaded into another application, and then runs the app. This is particularly useful in the case that your app might be used on its own or as part of another application:

```
if __name__ == '__main__':
    app.run()
```

Next, we'll need the templates for rendering html.

Let's make a templates folder for all the templates to live in:

```
mkdir templates
```

And create all the template files that we need:

```
touch templates/base.html templates/index.html templates/post.html
```

Add this content to the base.html file:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>{{ g.title }}</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>

<header>
  <div class="container">
    <h1><a href="/">{{ g.title }}</a></h1>
  </div>
</header>

<main role="main">
  <div class="container">
    {% block content %}{% endblock %}
  </main>
</div>

<footer>
  <div class="container">
    <p>Posts are also available via json at <a href="/api/posts">/api/posts</a>/
  </div>
</footer>

</body>
</html>
```

Add this content to the index.html file:

```
{% extends "base.html" %}

{% block content %}

{% for post in posts %}
<h3>
  <a href="/post/{{ post.slug }}">
    {{ post.title }}
  </a>
</h3>
<div>{{ post.content }}</div>
{% endfor %}

{% endblock %}
```

Add this content to the post.erb file:

```
{% extends "base.html" %}

{% block content %}

<h3>{{ post.title }}</h3>
<div>{{ post.content }}</div>

{% endblock %}
```

Let's add some css styling so this looks a little more readable. By default flask will look in a folder named static for static files.

First create the static folder and the styles.css file:

```
mkdir static
touch static/styles.css
```

Now add this content to the styles.css file:

```
body {
  font: 16px/1.5 'Helvetica Neue', Helvetica, Arial, sans-serif;
  color: #787876;
}

h1, h3 {
  font-weight: 300;
  margin-bottom: 5px;
}

a {
  text-decoration: none;
  color: #EA6045;
}

a:hover {
  color: #2F3440;
}

.container {
  width: 90%;
  margin: 0px auto;
}

footer {
  margin-top: 30px;
  border-top: 1px solid #efefef;
  padding-top: 20px;
  font-style: italic;
}

@media (min-width: 600px){
  .container {
    width: 60%;
  }
}
```

You should now be able to navigate on the home page, three blog post pages, and the posts json feed. Run the project with the nodemon command:

```
python app.py
```

Flask resources

Learn more about flask at the project website: http://flask.pocoo.org/[9]

Resources

Dive into Python: http://www.diveintopython.net/[10]

Test-driven development in python: http://net.tutsplus.com/tutorials/python-tutorials/test-driven-development-in-python/[11]

[9] http://flask.pocoo.org/
[10] http://www.diveintopython.net/
[11] http://net.tutsplus.com/tutorials/python-tutorials/test-driven-development-in-python/

12 Summary

Let's recap some of what we covered by comparing a few of the aspects of developing with ruby, javascript, and python.

Package managers & dependency management

Ruby:

Ruby developers use rubygems for installing packages, and use bundler & the Gemfile for defining dependencies.

Javascript & Node.js:

Javascript has npm for installing packages, and you can define dependencies. Remember that npm can be used for client-side code (particulary when using a tool like browserify), but you can also use package managers designed specifically for client-side code, like bower and component, which use bower.json and component.json as the files where you define dependencies.

Python:

Python has a few options for installing packages, but I recommend using pip. With pip you'll define dependencies in a requirements.txt file. It's also common to use pip in combination with a tool like virtualenv, which keeps the dependencies of your application separate from what's installed globally on your operating system.

Build tools / task automation

We took a quick look at npm scripts and Grunt for JavaScript, rake for Ruby, and fabric for Python, providing a glimpse of the differences in the way the build tools work with each language.

Simple web framework examples

Ruby:

```
require 'sinatra'

get '/' do
  'pizza is awesome'
end
```

Look at how tiny and pleasant that ruby code is!

Javascript:

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  res.send('pizza is awesome.');
});

app.listen(3000);

console.log('app is listening at localhost:3000');
```

Express doesn't automatically take care of listening on a default port, or telling the user that the app is listening, so that adds just a small amount of extra code compared to the ruby/sinatra example.

Python:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def pizza():
    return 'pizza is awesome'

if __name__ == '__main__':
    app.run()
```

Python feels different because of it's use of meaningful whitespace and lack of curly brackets or `do end` for blocks. Everything is just indented `4` spaces instead to represent blocks of code.

These three examples still feel very similar, though. That's no coincidence. Both express and flask were inspired by sinatra's clean and simple API.

13 Continued learning

Other books

There's a good chance that if you like this book you'll be interested in the other books in the Learn.js series.

Check them out!

- Introduction to JavaScript & Node.js[1]
- Making 2d Games with Node.js & Browserify[2]
- Mapping with Leaflet.js[3]
- Theming with Ghost[4]
- npm recipes[5]

Learn more at learnjs.io[6].

---

[1] http://learnjs.io/books/learnjs-01
[2] http://learnjs.io/books/learnjs-02
[3] http://learnjs.io/books/learnjs-03
[4] http://themingwithghost.com
[5] http://learnjs.io/npm-recipes
[6] http://learnjs.io

14 Changelog

v0.4.1 – February 28, 2014

- small typo fixes

v0.4.0 – February 26, 2014

- revise introduction
- expand 02-whatisdevenv
- improve windows instructions
- add to vagrant section
- update editors chapter
- add resources to git section
- change `DevEnvs` folder to `dev-envs`
- add npm scripts info to javascript automating tasks section
- update js import/require section
- add resources to javascript section
- the automating tasks sections of ruby & python chapters need actual content
- add to ending summary.
- fix typos and clean up resource link text styles

v0.3.0 - Feruary 3, 2014

- add sublime text editor tips and small typo edits/revisions
- add info about package control to sublime section
- fixes from suisea[1]
- update git and javascript chapters

v0.2.0 - October 26, 2013

- Expand terminal and vagrant sections
- Add vagrant instructions to ruby, javascript, and python chapters

[1] https://github.com/suisea

v0.1.4 - September 30, 2013

- Add ruby/sinatra extended example
- Add python/flask extended example
- fix some typos

v0.1.3 - September 27, 2013

- Add to unittest and other edits in python section
- Add extended express example to javascript section

v0.1.2 - September 3, 2013

- added rake, grunt, and fabric sections
- initial work on python language basics

v0.1.1 - August 18, 2013

- added language basics to javascript and ruby chapters
- added a bunch to vagrant chapter
- added to terminal chapter

v0.1.0 - August 17, 2013

- started all chapters!