



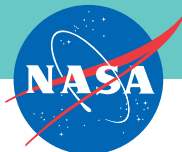
# Ansible Up & Running

AUTOMATING CONFIGURATION MANAGEMENT  
AND DEPLOYMENT THE EASY WAY

Lorin Hochstein



***“Ansible Tower has allowed us to provide better operations and security to our clients. It has also increased our efficiency as a team.”***



NASA uses Ansible Tower to centralize and control their Ansible automation initiative. With a real-time dashboard, role-based access control, credentials security, job scheduling, graphical inventory management and more, Ansible Tower is the best way to run Ansible in your organization, too.

Try Ansible Tower for free at **[ansible.com/tower](https://ansible.com/tower)**

# ANSIBLE

Copyright © 2014 ANSIBLE, INC. All rights reserved.

This Excerpt contains Chapters 2 and 13 of the book *Ansible: Up and Running*.  
The complete book is available at [oreilly.com](http://oreilly.com) and through other retailers.

---

# Ansible: Up and Running

*Lorin Hochstein*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

**O'REILLY®**

## **Ansible: Up and Running**

by Lorin Hochstein

Copyright © 2015 Lorin Hochstein. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editor:** Brian Anderson

**Production Editor:** Melanie Yarbrough

**Copyeditor:** Carla Thornton

**Proofreader:** Marta Justak

**Indexer:** WordCo Indexing Services

**Interior Designer:** David Futato

**Cover Designer:** Ellie Volkhausen

**Illustrator:** Rebecca Demarest

May 2015:

First Edition

### **Revision History for the First Edition**

2015-04-28: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491915325> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Ansible: Up and Running*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91532-5

[LSI]

---

# Table of Contents

<b>Foreword.....</b>	<b>v</b>
<b>Preface.....</b>	<b>vii</b>
<b>1. Playbooks: A Beginning.....</b>	<b>13</b>
Some Preliminaries	13
A Very Simple Playbook	14
Specifying an nginx Config File	16
Creating a Custom Homepage	17
Creating a Webservers Group	17
Running the Playbook	18
Playbooks Are YAML	20
Start of File	20
Comments	20
Strings	20
Booleans	21
Lists	21
Dictionaries	22
Line Folding	22
Anatomy of a Playbook	23
Plays	24
Tasks	25
Modules	26
Putting It All Together	27
Did Anything Change? Tracking Host State	28
Getting Fancier: TLS Support	28
Generating TLS certificate	30
Variables	30

Generating the Nginx Configuration Template	32
Handlers	33
Running the Playbook	35
<b>2. Docker.....</b>	<b>37</b>
The Case for Pairing Docker with Ansible	38
Docker Application Life Cycle	39
Dockerizing Our Mezzanine Application	40
Creating Docker Images with Ansible	42
Mezzanine	42
The Other Container Images	47
Postgres	47
Memcached	47
Nginx	48
Certs	49
Building the Images	50
Deploying the Dockerized Application	51
Starting the Database Container	51
Retrieving the Database Container IP Address and Mapped Port	52
Waiting for the Database to Start Up	56
Initializing the Database	57
Starting the Memcached Container	58
Starting the Mezzanine Container	58
Starting the Certificate Container	59
Starting the Nginx Container	59
The Entire Playbook	60

---

# Foreword

Ansible started as a simple side project in February of 2012, and its rapid growth has been a pleasant surprise. It is now the work product of about a thousand people (and the ideas of many more than that), and it is widely deployed in almost every country. It's not unusual in a computer meet-up to find a handful (at least) of people who use it.

Ansible is perhaps exciting because it really isn't. Ansible doesn't really attempt to break new ground, but rather to distill a lot of existing ideas that other smart folks had already figured out and make them a bit more accessible.

Ansible sought a middle ground between somewhat computer-sciencey IT automation approaches (themselves a reaction to tedious large commercial suites) and hack-and-slash scripting that just got things done. Also, how can we replace a configuration management system, a deployment project, an orchestration project, and our library of arbitrary but important shell scripts with a single system? That was the idea.

Could we remove major architectural components from the IT automation stack? Eliminating management demons and relying instead on OpenSSH meant the system could start managing a computer fleet immediately, without having to set up anything on the managed machines. Further, the system was apt to be more reliable and secure.

I had noticed that in trying to automate systems previously, things that should be simple were often hard, and that writing automation content could often create a time-sucking force that kept me from things I wanted to spend more time doing. And I didn't want the system to take months to become an expert with, either.

In particular, I personally enjoy writing new software, but piloting automation systems, a bit less. In short, I wanted to make automation quicker and leave me more time for the things I cared about. Ansible was not something you were meant to use all day long, but to get in, get out, and get back to doing the things you cared about.

I hope you will like Ansible for many of the same reasons.



Although I spent a large amount of time making sure Ansible's docs were comprehensive, there's always a strong advantage to seeing material presented in a variety of ways, and often in seeing actual practice applied alongside the reference material.

In *Ansible: Up And Running*, Lorin presents Ansible in a very idiomatic way, in exactly the right order in which you might wish to explore it. Lorin has been around Ansible since almost the very beginning, and I'm very grateful for his contributions and input.

I'm also immensely thankful for everyone who has been a part of this project to date, and everyone who will be in the future.

Enjoy the book, and enjoy managing your computer fleet! And remember to install cowsay!

— *Michael DeHaan*

— *Creator of Ansible (software), former  
CTO of Ansible, Inc. (company)*

— *April 2015*

## Why I Wrote This Book

When I was writing my first web application, using Django, the popular Python-based framework, I remember the sense of accomplishment when the app was finally working on my desktop. I would run `django manage.py runserver`, point my browser to <http://localhost:8000>, and there was my web application in all its glory.

Then I discovered there were all of these...*things* I had to do, just to get the darned app to run on the Linux server. In addition to installing Django and my app onto the server, I had to install Apache and the `mod_python` module so that Apache could run Django apps. Then I had to figure out the right Apache configuration file incantation so that it would run my application and serve up the static assets properly.

None of it was hard, it was just a pain to get all of those details right. I didn't want to muck about with configuration files, I just wanted my app to run. Once I got it working, everything was fine...until, several months later, I had to do it again, on a different server, at which point I had to start the process all over again.

Eventually, I discovered that this process was Doing It Wrong. The right way to do this sort of thing has a name, and that name is *configuration management*. The great thing about using configuration management is that it's a way to capture knowledge that always stays up-to-date. No more hunting for the right doc page or searching through your old notes.

Recently, a colleague at work was interested in trying out Ansible for deploying a new project, and he asked me for a reference on how to apply the Ansible concepts in practice, beyond what was available in the official docs. I didn't know what else to recommend, so I decided to write something to fill the gap—and here it is. Alas, this book comes too late for him, but I hope you'll find it useful.

# Who Should Read This Book

This book is for anyone who needs to deal with Linux or Unix-like servers. If you've ever used the terms *systems administration*, *operations*, *deployment*, *configuration management*, or (sigh) *DevOps*, then you should find some value here.

Although I have managed my share of Linux servers, my background is in software engineering. This means that the examples in this book tend toward the deployment end of the spectrum, although I'm in agreement with Andrew Clay Shafer (???) that the distinction between deployment and configuration is unresolved.

## Navigating This Book

I'm not a big fan of book outlines: Chapter 1 covers so and so, Chapter 2 covers such and such, that sort of thing. I strongly suspect that nobody ever reads them (I never do), and the table of contents is much easier to scan.

This book is written to be read start to finish, with later chapters building on the earlier ones. It's written largely in a tutorial style, so you should be able to follow along on your own machine. Most of the examples are focused on web applications.

## Conventions Used in This Book

The following typographical conventions are used in this book:

### *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

### Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

### Constant width bold

Shows commands or other text that should be typed literally by the user.

### *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

## Online Resources

Code samples from this book are available at this book's [GitHub page](#). There is ample official [Ansible documentation](#) available for reference.

I maintain a few Ansible quick reference pages on [GitHub](#) as well.

The Ansible code is on GitHub, split across three repositories:

- [Main repo](#)
- [Core modules](#)
- [Extra modules](#)

Bookmark the [Ansible module index](#); you'll be referring to it constantly as you use Ansible. [Ansible Galaxy](#) is a repository of Ansible roles contributed by the community. The [Ansible Project Google Group](#) is the place to go if you have any questions about Ansible.

If you're interested in contributing to Ansible development, check out the [Ansible Development Google Group](#).

For real-time help with Ansible, there's an active `#ansible` IRC channel on `irc.free-node.net`.


Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/lorin/ansiblebook>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Ansible: Up and Running* by Lorin Hochstein (O'Reilly). Copyright 2015 Lorin Hochstein, 978-1-491-91532-5.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Safari® Books Online

 **Safari**® *Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise, government, education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us **online**.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (in the United States or Canada)  
707-829-0515 (international or local)  
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/ansible-up-and-running>.

To comment or ask technical questions about this book, send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

## Acknowledgments

Thanks to Jan-Piet Mens, Matt Jaynes, and John Jarvis for reviewing drafts of the book and providing feedback. Thanks to Isaac Saldana and Mike Rowan at SendGrid for being so supportive of this endeavor. Thanks to Michael DeHaan for creating Ansible and shepherding the community that sprang up around it, as well as for providing feedback on the book, including an explanation of why he chose to use the name “Ansible.” Thanks to my editor, Brian Anderson, for his endless patience in working with me.

Thanks to Mom and Dad for their unfailing support; my brother Eric, the actual writer in the family; and my two sons, Benjamin and Julian. Finally, thanks to my wife, Stacy, for everything.



---

# Playbooks: A Beginning

Most of your time in Ansible will be spent writing *playbooks*. A playbook is the term that Ansible uses for a configuration management script. Let's look at an example: installing the nginx web server and configuring it for secure communication.

If you're following along in this chapter, you should end up with the files listed here:

- *playbooks/ansible.cfg*
- *playbooks/hosts*
- *playbooks/Vagrantfile*
- *playbooks/web-notls.yml*
- *playbooks/web-tls.yml*
- *playbooks/files/nginx.key*
- *playbooks/files/nginx.crt*
- *playbooks/files/nginx.conf*
- *playbooks/templates/index.html.j2*
- *playbooks/templates/nginx.conf.j2*

## Some Preliminaries

Before we can run this playbook against our Vagrant machine, we need to expose ports 80 and 443, so we can access them. As shown in **Figure 1-1**, we are going to configure Vagrant so that requests to ports 8080 and 8443 on our local machine are forwarded to ports 80 and 443 on the Vagrant machine. This will allow us to access



the web server running inside Vagrant at *http://localhost:8080* and *https://localhost:8443*.

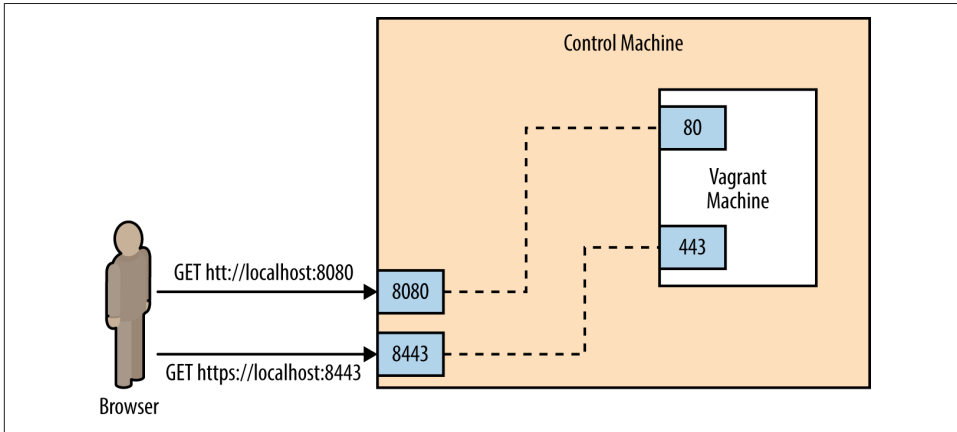


Figure 1-1. Exposing ports on Vagrant machine

Modify your *Vagrantfile* so it looks like this:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.network "forwarded_port", guest: 443, host: 8443
end
```

This maps port 8080 on your local machine to port 80 of the Vagrant machine, and port 8443 on your local machine to port 443 on the Vagrant machine. Once you make the changes, tell Vagrant to have them go into effect by running:

```
$ vagrant reload
```

You should see output that includes:

```
==> default: Forwarding ports...
      default: 80 => 8080 (adapter 1)
      default: 443 => 8443 (adapter 1)
      default: 22 => 2222 (adapter 1)
```

## A Very Simple Playbook

For our first example playbook, we'll configure a host to run an nginx web server. For this example, we won't configure the web server to support TLS encryption. This will make setting up the web server simpler, but a proper website should have TLS encryption enabled, and we'll cover how to do that later on in this chapter.

First, we'll see what happens when we run the playbook in [Example 1-1](#), and then we'll go over the contents of the playbook in detail.

### *Example 1-1. web-notls.yml*

```
- name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

## Why Do You Use “True” in One Place and “Yes” in Another?

Sharp-eyed readers might have noticed that [Example 1-1](#) uses `True` in one spot in the playbook (to enable `sudo`) and `yes` in another spot in the playbook (to update the apt cache).

Ansible is pretty flexible on how you represent truthy and falsey values in playbooks. Strictly speaking, module arguments (like `update_cache=yes`) are treated differently from values elsewhere in playbooks (like `sudo: True`). Values elsewhere are handled by the YAML parser and so use the YAML conventions of truthiness, which are:

### *YAML truthy*

`true`, `True`, `TRUE`, `yes`, `Yes`, `YES`, `on`, `On`, `ON`, `y`, `Y`

### *YAML falsey*

`false`, `False`, `FALSE`, `no`, `No`, `NO`, `off`, `Off`, `OFF`, `n`, `N`

Module arguments are passed as strings and use Ansible's internal conventions, which are:

```
module arg truthy
    yes, on, 1, true
```

```
module arg falsey
    no, off, 0, false
```

I tend to follow the examples in the official Ansible documentation. These typically use `yes` and `no` when passing arguments to modules (since that's consistent with the module documentation), and `True` and `False` elsewhere in playbooks.

## Specifying an nginx Config File

This playbook requires two additional files before we can run it. First, we need to define an nginx configuration file.

Nginx ships with a configuration file that works out of the box if you just want to serve static files. But you'll almost always need to customize this, so we'll overwrite the default configuration file with our own as part of this playbook. As we'll see later, we'll need to modify this configuration file to support TLS. **Example 1-2** shows a basic nginx config file. Put it in `playbooks/files/nginx.conf`.<sup>1</sup>

*Example 1-2. files/nginx.conf*

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root /usr/share/nginx/html;
    index index.html index.htm;

    server_name localhost;

    location / {
        try_files $uri $uri/ =404;
    }
}
```



An Ansible convention is to keep files in a subdirectory named *files* and Jinja2 templates in a subdirectory named *templates*. I'll follow this convention throughout the book.

---

<sup>1</sup> Note that while we call this file `nginx.conf`, it replaces the `sites-enabled/default` nginx server block config file, not the main `/etc/nginx.conf` config file.

## Creating a Custom Homepage

Let's add a custom homepage. We're going to use Ansible's template functionality so that Ansible will generate the file from a template. Put the file shown in [Example 1-3](#) in `playbooks/templates/index.html.j2`.

*Example 1-3. `playbooks/templates/index.html.j2`*

```
<html>
  <head>
    <title>Welcome to ansible</title>
  </head>
  <body>
    <h1>nginx, configured by Ansible</h1>
    <p>If you can see this, Ansible successfully installed nginx.</p>

    <p>{{ ansible_managed }}</p>
  </body>
</html>
```

This template references a special Ansible variable named `ansible_managed`. When Ansible renders this template, it will replace this variable with information about when the template file was generated. [Figure 1-2](#) shows a screenshot of a web browser viewing the generated HTML.



*Figure 1-2. Rendered HTML*

## Creating a Webservers Group

Let's create a "webservers" group in our inventory file so that we can refer to this group in our playbook. For now, this group will contain our testserver.

Inventory files are in the `.ini` file format. We'll go into this format in detail later in the book. Edit your `playbooks/hosts` file to put a `[webservers]` line above the `testserver` line, as shown in [Example 1-4](#). This indicates that `testserver` is in the `webservers` group.

### *Example 1-4. playbooks/hosts*

```
[webservers]
testserver ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
```

You should now be able to ping the webservers group using the ansible command-line tool:

```
$ ansible webservers -m ping
```

The output should look like this:

```
testserver | success >> {
    "changed": false,
    "ping": "pong"
}
```

## Running the Playbook

The `ansible-playbook` command executes playbooks. To run the playbook, do:

```
$ ansible-playbook web-notls.yml
```

**Example 1-5** shows what the output should look.

### *Example 1-5. Output of ansible-playbook*

```
PLAY [Configure webserver with nginx] *****

GATHERING FACTS *****
ok: [testserver]

TASK: [install nginx] *****
changed: [testserver]

TASK: [copy nginx config file] *****
changed: [testserver]

TASK: [enable configuration] *****
ok: [testserver]

TASK: [copy index.html] *****
changed: [testserver]

TASK: [restart nginx] *****
changed: [testserver]

PLAY RECAP *****
testserver          : ok=6    changed=4    unreachable=0    failed=0
```

## Cowsay

If you have the *cowsay* program installed on your local machine, then Ansible output will look like this instead:

```
< PLAY [Configure webserver with nginx] >
-----
      \   ^__^
       (oo)\_______
            (_____)  )\\/\
                 ||----w |
                 ||     ||
```

If you don't want to see the cows, you can disable cowsay by setting the `ANSIBLE_NOCOWS` environment variable like this:

```
$ export ANSIBLE_NOCOWS=1
```

You can also disable cowsay by adding the following to your *ansible.cfg* file.

```
[defaults]
nocows = 1
```

If you didn't get any errors,<sup>2</sup> you should be able to point your browser to <http://localhost:8080> and see the custom HTML page, as shown in [Figure 1-2](#).



If your playbook file is marked as executable and starts with a line that looks like this:<sup>3</sup>

```
#!/usr/bin/env ansible-playbook
```

then you can execute it by invoking it directly, like this:

```
$ ./web-notls.yml
```

<sup>2</sup> If you encountered an error, you might want to skip to ??? for assistance on debugging.

<sup>3</sup> Colloquially referred to as a “shebang.”

## What's This "Gathering Facts" Business?

You might have noticed the following lines of output when Ansible first starts to run:

```
GATHERING FACTS *****
ok: [testserver]
```

When Ansible starts executing a play, the first thing it does is collect information about the server it is connecting to, including which operating system is running, hostname, IP and MAC addresses of all interfaces, and so on.

You can then use this information later on in the playbook. For example, you might need the IP address of the machine for populating a configuration file.

You can turn off *fact gathering* if you don't need it, in order to save some time. We'll cover the use of facts and how to disable fact gathering in a later chapter.

## Playbooks Are YAML

Ansible playbooks are written in YAML syntax. YAML is a file format similar in intent to JSON, but generally easier for humans to read and write. Before we go over the playbook, let's cover the concepts of YAML that are most important for writing playbooks.

### Start of File

YAML files are supposed to start with three dashes to indicate the beginning of the document:

```
---
```

However, if you forget to put those three dashes at the top of your playbook files, Ansible won't complain.

### Comments

Comments start with a number sign and apply to the end of the line, the same as in shell scripts, Python, and Ruby:

```
# This is a YAML comment
```

### Strings

In general, YAML strings don't have to be quoted, although you can quote them if you prefer. Even if there are spaces, you don't need to quote them. For example, this is a string in YAML:

```
this is a lovely sentence
```

The JSON equivalent is:

```
"this is a lovely sentence"
```

There are some scenarios in Ansible where you will need to quote strings. These typically involve the use of `{{ braces }}` for variable substitution. We'll get to those later.

## Booleans

YAML has a native Boolean type, and provides you with a wide variety of strings that can be interpreted as true or false, which we covered in [“Why Do You Use “True” in One Place and “Yes” in Another?” on page 15](#).

Personally, I always use `True` and `False` in my Ansible playbooks.

For example, this is a Boolean in YAML:

```
True
```

The JSON equivalent is:

```
true
```

## Lists

YAML lists are like arrays in JSON and Ruby or lists in Python. Technically, these are called *sequences* in YAML, but I call them *lists* here to be consistent with the official Ansible documentation.

They are delimited with hyphens, like this:

```
- My Fair Lady
- Oklahoma
- The Pirates of Penzance
```

The JSON equivalent is:

```
[
  "My Fair Lady",
  "Oklahoma",
  "The Pirates of Penzance"
]
```

(Note again how we didn't have to quote the strings in YAML, even though they have spaces in them.)

YAML also supports an inline format for lists, which looks like this:

```
[My Fair Lady, Oklahoma, The Pirates of Penzance]
```



## Dictionaries

YAML *dictionaries* are like objects in JSON, dictionaries in Python, or hashes in Ruby. Technically, these are called *mappings* in YAML, but I call them *dictionaries* here to be consistent with the official Ansible documentation.

They look like this:

```
address: 742 Evergreen Terrace
city: Springfield
state: North Takoma
```

The JSON equivalent is:

```
{
  "address": "742 Evergreen Terrace",
  "city": "Springfield",
  "state": "North Takoma"
}
```

YAML also supports an inline format for dictionaries, which looks like this:

```
{address: 742 Evergreen Terrace, city: Springfield, state: North Takoma}
```

## Line Folding

When writing playbooks, you'll often encounter situations where you're passing many arguments to a module. For aesthetics, you might want to break this up across multiple lines in your file, but you want Ansible to treat the string as if it were a single line.

You can do this with YAML using line folding with the greater than (>) character. The YAML parser will replace line breaks with spaces. For example:

```
address: >
  Department of Computer Science,
  A.V. Williams Building,
  University of Maryland
city: College Park
state: Maryland
```

The JSON equivalent is:

```
{
  "address": "Department of Computer Science, A.V. Williams Building,
  University of Maryland",
  "city": "College Park",
  "state": "Maryland"
}
```

# Anatomy of a Playbook

Let's take a look at our playbook from the perspective of a YAML file. Here it is again, in [Example 1-6](#).

*Example 1-6. web-notls.yml*

```
- name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

In [Example 1-7](#), we see the JSON equivalent of this file.

*Example 1-7. JSON equivalent of web-notls.yml*

```
[
  {
    "name": "Configure webserver with nginx",
    "hosts": "webservers",
    "sudo": true,
    "tasks": [
      {
        "name": "Install nginx",
        "apt": "name=nginx update_cache=yes"
      },
      {
        "name": "copy nginx config file",
        "template": "src=files/nginx.conf dest=/etc/nginx/
          sites-available/default"
      },
    ]
  }
]
```

```

    "name": "enable configuration",
    "file": "dest=/etc/nginx/sites-enabled/default src=/etc/nginx/sites-available
/default state=link"
  },
  {
    "name": "copy index.html",
    "template" : "src=templates/index.html.j2 dest=/usr/share/nginx/html/
index.html mode=0644"
  },
  {
    "name": "restart nginx",
    "service": "name=nginx state=restarted"
  }
]
}
]

```



A valid JSON file is also a valid YAML file. This is because YAML allows strings to be quoted, considers `true` and `false` to be valid Booleans, and has inline lists and dictionary syntaxes that are the same as JSON arrays and objects. But don't write your playbooks as JSON—the whole point of YAML is that it's easier for people to read.

## Plays

Looking at either the YAML or JSON representation, it should be clear that a playbook is a list of dictionaries. Specifically, a playbook is a list of *plays*.

Here's the play<sup>4</sup> from our example:

```

- name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

```

---

<sup>4</sup> Actually, it's a list that contains a single play.

```
- name: copy index.html
  template: src=templates/index.html.j2
           dest=/usr/share/nginx/html/index.html mode=0644

- name: restart nginx
  service: name=nginx state=restarted
```

Every play must contain:

- A set of *hosts* to configure
- A list of *tasks* to be executed on those hosts

Think of a play as the thing that connects hosts to tasks.

In addition to specifying hosts and tasks, plays also support a number of optional settings. We'll get into those later, but three common ones are:

**name**

A comment that describes what the play is about. Ansible will print this out when the play starts to run.

**sudo**

If true, Ansible will run every task by sudo'ing as (by default) the root user. This is useful when managing Ubuntu servers, since by default you cannot SSH as the root user.

**vars**

A list of variables and values. We'll see this in action later in this chapter.

## Tasks

Our example playbook contains one play that has five tasks. Here's the first task of that play:

```
- name: install nginx
  apt: name=nginx update_cache=yes
```

The name is optional, so it's perfectly valid to write a task like this:

```
- apt: name=nginx update_cache=yes
```

Even though names are optional, I recommend you use them because they serve as good reminders for the intent of the task. (Names will be very useful when somebody else is trying to understand your playbook, including yourself in six months.) As we've seen, Ansible will print out the name of a task when it runs. Finally, as we'll see in [???](#), you can use the `--start-at-task <task name>` flag to tell ansible-playbook to start a playbook in the middle of a task, but you need to reference the task by name.

Every task must contain a key with the name of a module and a value with the arguments to that module. In the preceding example, the module name is *apt* and the arguments are `name=nginx` `update_cache=yes`.

These arguments tell the *apt* module to install the package named *nginx* and to update the package cache (the equivalent of doing an `apt-get update`) before installing the package.

It's important to understand that, from the point of the view of the YAML parser used by the Ansible frontend, the arguments are treated as a string, not as a dictionary. This means that if you want to break up arguments into multiple lines, you need to use the YAML folding syntax, like this:

```
- name: install nginx
  apt: >
    name=nginx
    update_cache=yes
```

Ansible also supports a task syntax that will let you specify module arguments as a YAML dictionary, which is helpful when using modules that support complex arguments. We'll cover that in ???.

Ansible also supports an older syntax that uses *action* as the key and puts the name of the module in the value. The preceding example also can be written as:

```
- name: install nginx
  action: apt name=nginx update_cache=yes
```

## Modules

Modules are scripts<sup>5</sup> that come packaged with Ansible and perform some kind of action on a host. Admittedly, that's a pretty generic description, but there's enormous variety across Ansible modules. The modules we use in this chapter are:

*apt*

Installs or removes packages using the *apt* package manager.

*copy*

Copies a file from local machine to the hosts.

*file*

Sets the attribute of a file, symlink, or directory.

*service*

Starts, stops, or restarts a service.

---

<sup>5</sup> The modules that ship with Ansible all are written in Python, but modules can be written in any language.

*template*

Generates a file from a template and copies it to the hosts.

## Viewing Ansible Module Documentation

Ansible ships with the `ansible-doc` command-line tool, which shows documentation about modules. Think of it as man pages for Ansible modules. For example, to show the documentation for the *service* module, run:

```
$ ansible-doc service
```

If you use Mac OS X, there's a wonderful documentation viewer called **Dash** that has support for Ansible. Dash indexes all of the Ansible module documentation. It's a commercial tool (\$19.99 as of this writing), but I find it invaluable.

Recall from the first chapter that Ansible executes a task on a host by generating a custom script based on the module name and arguments, and then copies this script to the host and runs it.

There are over 200 modules that ship with Ansible, and this number grows with every release. You can also find third-party Ansible modules out there, or write your own.

## Putting It All Together

To sum up, a playbook contains one or more plays. A play associates an unordered set of hosts with an ordered list of task\_. Each task is associated with exactly one module.

**Figure 1-3** is an entity-relationship diagram that depicts this relationship between playbooks, plays, hosts, tasks, and modules.

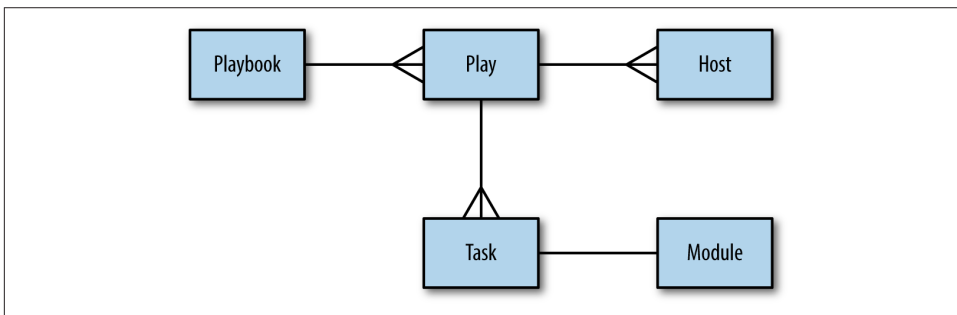


Figure 1-3. Entity-relationship diagram

## Did Anything Change? Tracking Host State

When you run `ansible-playbook`, Ansible outputs status information for each task it executes in the play.

Looking back at [Example 1-5](#), notice that the status for some of the tasks is *changed*, and the status for some others is *ok*. For example, the `install nginx` task has status *changed*, which appears as yellow on my terminal.

```
TASK: [install nginx] *****
changed: [testserver]
```

The `enable configuration`, on the other hand, has status *ok*, which appears as green on my terminal:

```
TASK: [enable configuration] *****
ok: [testserver]
```

Any Ansible task that runs has the potential to change the state of the host in some way. Ansible modules will first check to see if the state of the host needs to be changed before taking any action. If the state of the host matches the arguments of the module, then Ansible takes no action on the host and responds with a state of *ok*.

On the other hand, if there is a difference between the state of the host and the arguments to the module, then Ansible will change the state of the host and return *changed*.

In the example output just shown, the `install nginx` task was changed, which meant that before I ran the playbook, the `nginx` package had not previously been installed on the host. The `enable configuration` task was unchanged, which meant that there was already a configuration file on the server that was identical to the file I was copying over. The reason for this is that the `nginx.conf` file I used in my playbook is the same as the `nginx.conf` file that gets installed by the `nginx` package on Ubuntu.

As we'll see later in this chapter, Ansible's detection of state change can be used to trigger additional actions through the use of *handlers*. But, even without using handlers, it is still a useful form of feedback to see whether your hosts are changing state as the playbook runs.

## Getting Fancier: TLS Support

Let's move on to a more complex example: We're going to modify the previous playbook so that our web servers support TLS. The new features here are:

- Variables
- Handlers

## TLS versus SSL

You might be familiar with the term *SSL* rather than *TLS* in the context of secure web servers. SSL is an older protocol that was used to secure communications between browsers and web servers, and it has been superseded by a newer protocol named TLS.

Although many continue to use the term *SSL* to refer to the current secure protocol, in this book, I use the more accurate *TLS*.

**Example 1-8** shows what our playbook looks like with TLS support.

*Example 1-8. web-tls.yml*

```
- name: Configure webserver with nginx and tls
  hosts: webservers
  sudo: True
  vars:
    key_file: /etc/nginx/ssl/nginx.key
    cert_file: /etc/nginx/ssl/nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost
  tasks:
    - name: Install nginx
      apt: name=nginx update_cache=yes cache_valid_time=3600

    - name: create directories for ssl certificates
      file: path=/etc/nginx/ssl state=directory

    - name: copy TLS key
      copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
      notify: restart nginx

    - name: copy TLS certificate
      copy: src=files/nginx.crt dest={{ cert_file }}
      notify: restart nginx

    - name: copy nginx config file
      template: src=templates/nginx.conf.j2 dest={{ conf_file }}
      notify: restart nginx

    - name: enable configuration
      file: dest=/etc/nginx/sites-enabled/default src={{ conf_file }} state=link
      notify: restart nginx

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
      mode=0644
```



```
handlers:
  - name: restart nginx
    service: name=nginx state=restarted
```

## Generating TLS certificate

We need to manually generate a TLS certificate. In a production environment, you'd purchase your TLS certificate from a certificate authority. We'll use a self-signed certificate, since we can generate those for free.

Create a *files* subdirectory of your *playbooks* directory, and then generate the TLS certificate and key:

```
$ mkdir files
$ openssl req -x509 -nodes -days 3650 -newkey rsa:2048 \
  -subj /CN=localhost \
  -keyout files/nginx.key -out files/nginx.crt
```

It should generate the files *nginx.key* and *nginx.crt* in the *files* directory. The certificate has an expiration date of 10 years (3,650 days) from the day you created it.

## Variables

The play in our playbook now has a section called *vars*:

```
vars:
  key_file: /etc/nginx/ssl/nginx.key
  cert_file: /etc/nginx/ssl/nginx.crt
  conf_file: /etc/nginx/sites-available/default
  server_name: localhost
```

This section defines four variables and assigns a value to each variable.

In our example, each value is a string (e.g., */etc/nginx/ssl/nginx.key*), but any valid YAML can be used as the value of a variable. You can use lists and dictionaries in addition to strings and Booleans.

Variables can be used in tasks, as well as in template files. You reference variables using the `{{ braces }}` notation. Ansible will replace these braces with the value of the variable.

Consider this task in the playbook:

```
- name: copy TLS key
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
```

Ansible will substitute `{{ key_file }}` with */etc/nginx/ssl/nginx.key* when it executes this task.

## When Quoting Is Necessary

If you reference a variable right after specifying the module, the YAML parser will misinterpret the variable reference as the beginning of an in-line dictionary. Consider the following example:

```
- name: perform some task
  command: {{ myapp }} -a foo
```

Ansible will try to parse the first part of `{{ myapp }} -a foo` as a dictionary instead of a string, and will return an error. In this case, you must quote the arguments:

```
- name: perform some task
  command: "{{ myapp }} -a foo"
```

A similar problem arises if your argument contains a colon. For example:

```
- name: show a debug message
  debug: msg="The debug module will print a message: neat, eh?"
```

The colon in the `msg` argument trips up the YAML parser. To get around this, you need to quote the entire argument string.

Unfortunately, just quoting the argument string won't resolve the problem, either.

```
- name: show a debug message
  debug: "msg=The debug module will print a message: neat, eh?"
```

This will make the YAML parser happy, but the output isn't what you expect:

```
TASK: [show a debug message] *****
ok: [localhost] => {
  "msg": "The"
}
```

The debug module's `msg` argument requires a quoted string to capture the spaces. In this particular case, we need to quote both the whole argument string and the `msg` argument. Ansible supports alternating single and double quotes, so you can do this:

```
- name: show a debug message
  debug: "msg='The debug module will print a message: neat, eh?'"
```

This yields the expected output:

```
TASK: [show a debug message] *****
ok: [localhost] => {
  "msg": "The debug module will print a message: neat, eh?"
}
```

Ansible is pretty good at generating meaningful error messages if you forget to put quotes in the right places and end up with invalid YAML.

## Generating the Nginx Configuration Template

If you’ve done web programming, you’ve likely used a template system to generate HTML. In case you haven’t, a template is just a text file that has some special syntax for specifying variables that should be replaced by values. If you’ve ever received an automated email from a company, they’re probably using an email template as shown in [Example 1-9](#).

*Example 1-9. An email template*

```
Dear {{ name }},  
  
You have {{ num_comments }} new comments on your blog: {{ blog_name }}.
```

Ansible’s use case isn’t HTML pages or emails—it’s configuration files. You don’t want to hand-edit configuration files if you can avoid it. This is especially true if you have to reuse the same bits of configuration data (say, the IP address of your queue server or your database credentials) across multiple configuration files. It’s much better to take the info that’s specific to your deployment, record it in one location, and then generate all of the files that need this information from templates.

Ansible uses the Jinja2 template engine to implement templating. If you’ve ever used a templating library such as Mustache, ERB, or the Django template system, Jinja2 will feel very familiar.

Nginx’s configuration file needs information about where to find the TLS key and certificate. We’re going to use Ansible’s templating functionality to define this configuration file so that we can avoid hard-coding values that might change.

In your *playbooks* directory, create a *templates* subdirectory and create the file *templates/nginx.conf.j2*, as shown in [Example 1-10](#).

*Example 1-10. templates/nginx.conf.j2*

```
server {  
    listen 80 default_server;  
    listen [::]:80 default_server ipv6only=on;  
  
    listen 443 ssl;  
  
    root /usr/share/nginx/html;  
    index index.html index.htm;  
  
    server_name {{ server_name }};  
    ssl_certificate {{ cert_file }};  
    ssl_certificate_key {{ key_file }};  
  
    location / {
```

```

        try_files $uri $uri/ =404;
    }
}

```

We use the `.j2` extension to indicate that the file is a Jinja2 template. However, you can use a different extension if you like; Ansible doesn't care.

In our template, we reference three variables:

`server_name`

The hostname of the web server (e.g., `www.example.com`)

`cert_file`

The path to the TLS certificate

`key_file`

The path to the TLS private key

We define these variables in the playbook.

Ansible also uses the Jinja2 template engine to evaluate variables in playbooks. Recall that we saw the `{{ conf_file }}` syntax in the playbook itself.



Early versions of Ansible used a dollar sign (\$) to do variable interpolation in playbooks instead of the braces. You used to dereference variable *foo* by writing `$foo`, where now you write `{{ foo }}`. The dollar sign syntax has been deprecated; if you encounter it in an example playbook you find on the Internet, then you're looking at older Ansible code.

You can use all of the Jinja2 features in your templates, but we won't cover them in detail here. Check out the [Jinja2 Template Designer Documentation](#) for more details. You probably won't need to use those advanced templating features, though. One Jinja2 feature you probably will use with Ansible is filters; we'll cover those in a later chapter.

## Handlers

Looking back at our `web-tls.yml` playbook, note that there are two new playbook elements we haven't discussed yet. There's a `handlers` section that looks like this:

```

handlers:
- name: restart nginx
  service: name=nginx state=restarted

```

In addition, several of the tasks contain a `notify` key. For example:

```
- name: copy TLS key
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
  notify: restart nginx
```

Handlers are one of the conditional forms that Ansible supports. A handler is similar to a task, but it runs only if it has been notified by a task. A task will fire the notification if Ansible recognizes that the task has changed the state of the system.

A task notifies a handler by passing the handler's name as the argument. In the preceding example, the handler's name is `restart nginx`. For an nginx server, we'd need to restart it<sup>6</sup> if any of the following happens:

- The TLS key changes
- The TLS certificate changes
- The configuration file changes
- The contents of the *sites-enabled* directory change

We put a `notify` statement on each of the tasks to ensure that Ansible restarts nginx if any of these conditions are met.

### A few things to keep in mind about handlers

Handlers only run after all of the tasks are run, and they only run once, even if they are notified multiple times. They always run in the order that they appear in the play, not the notification order.

The official Ansible docs mention that the only common uses for handlers are for restarting services and for reboots. Personally, I've only ever used them for restarting services. Even then, it's a pretty small optimization, since we can always just unconditionally restart the service at the end of the playbook instead of notifying it on change, and restarting a service doesn't usually take very long.

Another pitfall with handlers that I've encountered is that they can be troublesome when debugging a playbook. It goes something like this:

1. I run a playbook.
2. One of my tasks with a *notify* on it changes state.
3. An error occurs on a subsequent task, stopping Ansible.
4. I fix the error in my playbook.
5. I run Ansible again.

---

<sup>6</sup> Alternatively, we could reload the configuration file using `state=reloaded` instead of restarting the service.

6. None of the tasks report a state change the second time around, so Ansible doesn't run the handler.

## Running the Playbook

As before, we use the `ansible-playbook` command to run the playbook.

```
$ ansible-playbook web-tls.yml
```

The output should look something like this:

```
PLAY [Configure webserver with nginx and tls] *****

GATHERING FACTS *****
ok: [testserver]

TASK: [Install nginx] *****
changed: [testserver]

TASK: [create directories for tls certificates] *****
changed: [testserver]

TASK: [copy TLS key] *****
changed: [testserver]

TASK: [copy TLS certificate] *****
changed: [testserver]

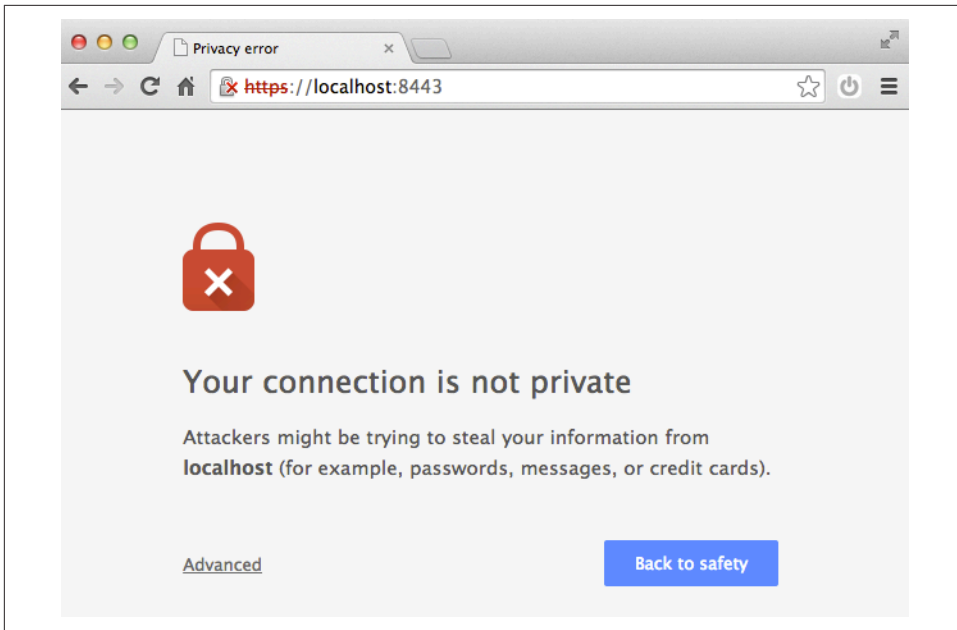
TASK: [copy nginx config file] *****
changed: [testserver]

TASK: [enable configuration] *****
ok: [testserver]

NOTIFIED: [restart nginx] *****
changed: [testserver]

PLAY RECAP *****
testserver                : ok=8    changed=6    unreachable=0    failed=0
```

Point your browser to <https://localhost:8443> (don't forget the "s" on https). If you're using Chrome, like I am, you'll get a ghastly message that says something like, "Your connection is not private" (see [Figure 1-4](#)).



*Figure 1-4. Browsers like Chrome don't trust self-signed TLS certificates*

Don't worry, though; that error is expected, as we generated a self-signed TLS certificate, and web browsers like Chrome only trust certificates that have been issued from a proper authority.

We covered a lot of the “what” of Ansible in this chapter, describing what Ansible will do to your hosts. The handlers we discussed here are just one form of control flow that Ansible supports. In a later chapter, we'll see iteration and conditionally running tasks based on the values of variables.

In the next chapter, we'll talk about the “who”; in other words, how to describe the hosts that your playbooks will run against.

The Docker project has taken the IT world by storm. I can't think of another technology that was so quickly embraced by the community. This chapter covers how to use Ansible to create Docker images and deploy Docker containers.

### What Is a Container?

A container is a form of virtualization. When you use virtualization to run processes in a guest operating system, these guest processes have no visibility into the host operating system that runs on the physical hardware. In particular, processes running in the guest are not able to directly access physical resources, even if these guest processes are provided with the illusion that they have root access.

Containers are sometimes referred to as *operating system virtualization* to distinguish them from *hardware virtualization* technologies.

In hardware virtualization, a program called the *hypervisor* virtualizes an entire physical machine, including a virtualized CPU, memory, and devices such as disks and network interfaces. Because the entire machine is virtualized, hardware virtualization is very flexible. In particular, you can run an entirely different operating system in the guest than in the host (e.g., running a Windows Server 2012 guest inside of a RedHat Enterprise Linux host), and you can suspend and resume a virtual machine just like you can a physical machine. This flexibility brings with it additional overhead needed to virtualize the hardware.

With operating system virtualization (containers), the guest processes are isolated from the host by the operating system. The guest processes run on the same kernel as the host. The host operating system is responsible for ensuring that the guest processes are fully isolated from the host. When running a Linux-based container program like Docker, the guest processes also must be Linux programs. However, the overhead is much lower than that of hardware virtualization, because you are running



only a single operating system. In particular, processes start up much more quickly inside containers than inside virtual machines.

Docker is more than just containers. Think of Docker as being a platform where containers are a building block. To use an analogy, containers are to Docker what virtual machines are to IaaS clouds. The other two major pieces that make up Docker are its image format and the Docker API.

You can think of Docker images as similar to virtual machine images. A Docker image contains a filesystem with an installed operating system, along with some metadata. One important difference is that Docker images are layered. You create a new Docker image by taking an existing Docker image and modifying it by adding, modifying, and deleting files. The representation for the new Docker image contains a reference to the original Docker image, as well as the file system differences between the original Docker image and the new Docker image. As an example, the official **nginx docker image** is built as layers on top of the official Debian Wheezy image. The layered approach means that Docker images are smaller than traditional virtual machine images, so it's faster to transfer Docker images over the Internet than it would be to transfer a traditional virtual machine image. The Docker project maintains a registry of publicly **available images**.

Docker also supports a remote API, which enables third-party tools to interact with it. In particular, Ansible's docker module uses the Docker remote API.

## The Case for Pairing Docker with Ansible

Docker containers make it easier to package your application into a single image that's easy to deploy in different places, which is why the Docker project has embraced the metaphor of the shipping container. Docker's remote API simplifies the automation of software systems that run on top of Docker.

There are two areas where Ansible simplifies working with Docker. One is in the orchestration of Docker containers. When you deploy a “Dockerized” software app, you're typically creating multiple Docker containers that contain different services. These services need to communicate with each other, so you need to connect the appropriate containers correctly and ensure they start up in the right order. Initially, the Docker project did not provide orchestration tools, so third-party tools emerged to fill in the gap. Ansible was built for doing orchestration, so it's a natural fit for deploying your Docker-based application.

The other area is the creation of Docker images. The official way to create your own Docker images is by writing special text files called *Dockerfiles*, which resemble shell scripts. For simpler images, Dockerfiles work just fine. However, when you start to

create more-complex images, you'll quickly miss the power that Ansible provides. Fortunately, you can use Ansible to create playbooks.

## Docker Application Life Cycle

Here's what the typical life cycle of a Docker-based application looks like:

1. Create Docker images on your local machine.
2. Push Docker images up from your local machine to the registry.
3. Pull Docker images down to your remote hosts from the registry.
4. Start up Docker containers on the remote hosts, passing in any configuration information to the containers on startup.

You typically create your Docker image on your local machine, or on a continuous integration system that supports creating Docker images, such as Jenkins or CircleCI. Once you've created your image, you need to store it somewhere it will be convenient for downloading onto your remote hosts.

Docker images typically reside in a repository called a *registry*. The Docker project runs a registry called *Docker Hub*, which can host both public and private Docker images, and where the Docker command-line tools have built-in support for pushing images up to a registry and for pulling images down from a registry.

Once your Docker image is in the registry, you connect to a remote host, pull down the container image, and then run the container. Note that if you try to run a container whose image isn't on the host, Docker will automatically pull down the image from the registry, so you do not need to explicitly issue a command to download an image from the registry.

When you use Ansible to create the Docker images and start the containers on the remote hosts, the application lifecycle looks like this:

1. Write Ansible playbooks for creating Docker images.
2. Run the playbooks to create Docker images on your local machine.
3. Push Docker images up from your local machine to the registry.
4. Write Ansible playbooks to pull Docker images down to remote hosts and start up Docker containers on remote hosts, passing in configuration information.
5. Run Ansible playbooks to start up the containers.

# Dockerizing Our Mezzanine Application

We'll use our Mezzanine example and deploy it inside of Docker containers. Recall that our application involves the following services:

- Postgres database
- Mezzanine (web application)
- Memcached (in-memory cache to improve performance)
- nginx (web server)

We could deploy all of these services into the same container. However, for pedagogical purposes, I'm going to run each service in a separate container, as shown in **Figure 2-1**. Deploying each service in a separate container makes for a more complex deployment, but it allows me to demonstrate how you can do more complex things with Docker and Ansible.

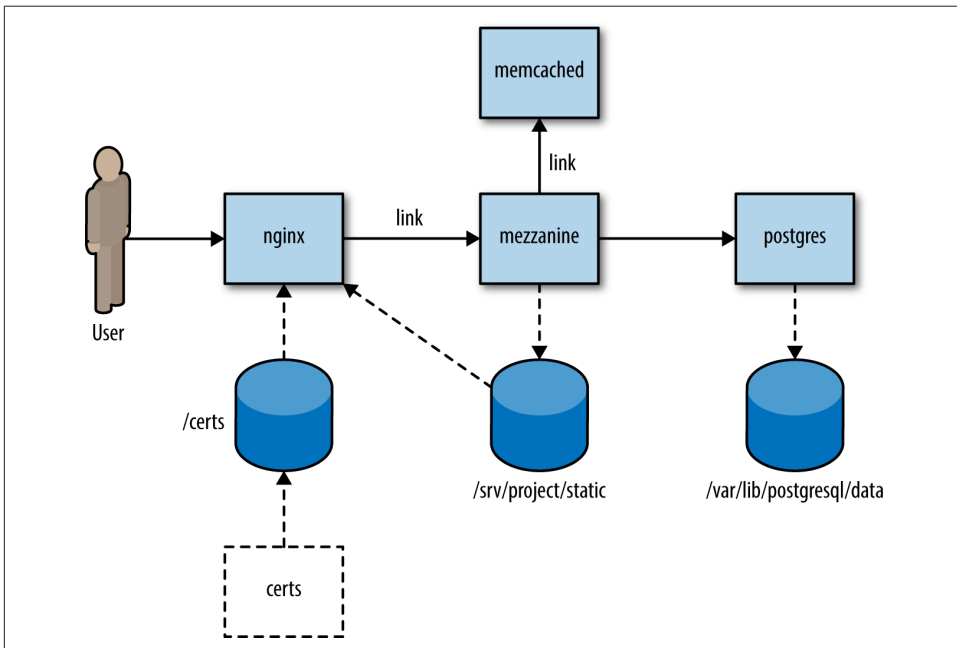


Figure 2-1. Deploying Mezzanine as Docker containers

Each box represents a Docker container that runs a service. Containers that communicate with each other using TCP/IP are connected by solid lines. The nginx container is the only one that must respond to requests from the outside world. It proxies web requests to the Mezzanine application, so it connects to the Mezzanine container. The Mezzanine container must access the database, so it connects to the Postgres

container. It must also connect to the Memcached container in order to access the in-memory cache provided by Memcached to improve performance.

The cylinders represent Docker volumes that containers export and import. For example, the Mezzanine container exports the volume `/srv/project/static`, and the nginx container imports this volume.

The nginx service must serve static content such as JavaScript, CSS, and images, including files uploaded by Mezzanine users. (Recall that Mezzanine is a CMS that allows users to upload files such as images.) These files are in the Mezzanine container, not the nginx container. To share files across these containers, we configure the Mezzanine container to store the static file content in a volume, and we mount the volume into the nginx container.

Containers that share volumes (in our deployment, *nginx* and *mezzanine*) must be running on the same host, but otherwise we could deploy each container on a separate host. In a real deployment, we'd likely deploy Memcached on the same host as Mezzanine, and we'd put Postgres on a separate host. In our example, I'm going to use *container linking* (see “[Linking Docker Containers](#)” on page 41) to link the nginx, Mezzanine, and Memcached containers together (hence the *link* annotation on the diagram). Mezzanine will communicate with Postgres over the port exposed by the Postgres container, in order to demonstrate both ways of connecting together containers that run on the same host.

## Linking Docker Containers

If two Docker containers are running on the same host, you can use a feature called *linking containers* so that the two containers can be networked together. Linking is unidirectional, so if container *A* is linked to container *B*, then processes in *A* can connect to network services running in *B*.

Docker will inject special environment variables into one of the containers. These variables contain IP addresses and ports so that one container can access services in the other container, as well as update the `/etc/hosts` file so that one container can access the other by hostname. For more details, see the official [Docker documentation about container](#).

Finally, there's a dashed box in the diagram labeled “certs.” This is a Docker data volume that contains the TLS certificates. Unlike the other containers, this one is stopped; it exists only to store the certificate files.

# Creating Docker Images with Ansible

In this chapter, I'm going to use the method recommended by the Ansible project for creating images with Ansible. In a nutshell, the method is:

1. Use an official Ansible base image that has Ansible installed in it.
2. In the Dockerfile, copy the playbooks into the image.
3. Invoke Ansible from the Dockerfile.

Note that we won't be creating all of our images with Ansible. In one case, we'll be able to use an existing image right off-the-shelf...er...Docker registry. In other cases, we'll build the Docker image with a traditional Dockerfile.

We need to create Docker images for each of the boxes depicted in [Figure 2-1](#).

## Mezzanine

Our Mezzanine container image is the most complex one, and we'll be using Ansible to configure it.

The official Ansible base images are hosted on the [Docker registry](#). As of this writing, there are two base images available:

- `ansible/centos7-ansible` (CentOS 7)
- `ansible/ubuntu14.04-ansible` (Ubuntu 14.04).

We'll be using the Ubuntu 14.04 image. To create this image, I have a *mezzanine* directory that contains the following files:

- *Dockerfile*
- *ansible/mezzanine-container.yml*
- *ansible/files/gunicorn.conf.py*
- *ansible/files/local\_settings.py*
- *ansible/files/scripts/setadmin.py*
- *ansible/files/scripts/setsite.py*

There's the Dockerfile for building the Docker image, the playbook itself (*mezzanine-container.yml*), and several other files that we're going to copy into the image.

[Example 2-1](#) shows what the Dockerfile looks like for building the Mezzanine image.

### Example 2-1. Mezzanine Dockerfile

```
FROM ansible/ubuntu14.04-ansible:stable
MAINTAINER Lorin Hochstein <lorin@ansiblebook.com>

ADD ansible /srv/ansible
WORKDIR /srv/ansible

RUN ansible-playbook mezzanine-container.yml -c local

VOLUME /srv/project/static

WORKDIR /srv/project

EXPOSE 8000
CMD ["gunicorn_django", "-c", "gunicorn.conf.py"]
```

We copy the playbook and associated files into the container and then execute the playbook. We also create a mount point for `/srv/project/static`, the directory that contains the static content that the nginx container will serve.

Finally, we expose port 8000 and specify `gunicorn_django` as the default command for the container, which will run Mezzanine using the Gunicorn application server. [Example 2-2](#) shows the playbook we use to configure the container.

### Example 2-2. `mezzanine-container.yml`

```
- name: Create Mezzanine container
  hosts: local
  vars:
    mezzanine_repo_url: https://github.com/lorin/mezzanine-example.git
    mezzanine_proj_path: /srv/project
    mezzanine_reqs_path: requirements.txt
    script_path: /srv/scripts
  tasks:
    - name: install apt packages
      apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
      with_items:
        - git
        - gunicorn
        - libjpeg-dev
        - libpq-dev
        - python-dev
        - python-pip
        - python-psycpg2
        - python-setuptools

    - name: check out the repository on the host
      git:
        repo: "{{ mezzanine_repo_url }}"
        dest: "{{ mezzanine_proj_path }}"
```

```

    accept_hostkey: yes

- name: install required python packages
  pip: name={{ item }}
  with_items:
    - south
    - psychopg2
    - django-compressor
    - python-memcached

- name: install requirements.txt
  pip: requirements={{ mezzanine_proj_path }}/{{ mezzanine_reqs_path }}

- name: generate the settings file
  copy: src=files/local_settings.py dest={{ mezzanine_proj_path }}/
    local_settings.py

- name: set the gunicorn config file
  copy: src=files/gunicorn.conf.py dest={{ mezzanine_proj_path }}/gunicorn.conf.py

- name: collect static assets into the appropriate directory
  django_manage: command=collectstatic app_path={{ mezzanine_proj_path }}
  environment:
    # We can't run collectstatic if the secret key is blank,
    # so we just pass in an arbitrary one
    SECRET_KEY: nonblanksecretkey

- name: script directory
  file: path={{ script_path }} state=directory

- name: copy scripts for setting site id and admin at launch time
  copy: src=files/scripts/{{ item }} dest={{ script_path }}/{{ item }} mode=0755
  with_items:
    - setadmin.py
    - setsite.py

```

The **Example 2-2** playbook is similar to the playbook from **???**, with the following differences:

- We don't install Postgres, nginx, Memcached, or Supervisor, which is discussed in **???**, into the image.
- We don't use templates to generate *local\_settings.py* and *gunicorn.conf.py*.
- We don't run the Django syncdb or migrate commands.
- We copy *setadmin.py* and *setsite.py* scripts into the container instead of executing them.

We don't install the other services into the image because those services are implemented by separate images, except for Supervisor.

## Why We Don't Need Supervisor

Recall that our deployment of Mezzanine originally used Supervisor to manage our application server (Gunicorn). This meant that Supervisor was responsible for starting and stopping the Gunicorn process.

In our Mezzanine Docker container, we don't need a separate program for starting and stopping the Gunicorn process. That's because Docker is itself a system designed for starting and stopping processes.

Without Docker, we would use Supervisor to start Gunicorn:

```
$ supervisorctl start gunicorn_mezzanine
```

With Docker, we start up a container containing Gunicorn, and we use Ansible to do something like this:

```
$ docker run lorin/mezzanine:latest
```

We don't use templates to generate *local\_settings.py* because when we build the image, we don't know what the settings will be. For example, we don't know what the database host, port, username, and password values should be. Even if we did, we don't want to hardcode them in the image, because we want to be able to use the same image in our development, staging, and production environments.

What we need is a service discovery mechanism so that we can determine what all of these settings should be when the container starts up. There are many different ways of implementing service discovery, including using a service discovery tool such as etcd, Consul, Apache ZooKeeper, or Eureka. We're going to use environment variables, since Docker lets us specify environment variables when we start containers.

**Example 2-3** shows the *local\_settings.py* file we are using for the image.

*Example 2-3. local\_settings.py*

```
from __future__ import unicode_literals
import os

SECRET_KEY = os.environ.get("SECRET_KEY", "")
NEVERCACHE_KEY = os.environ.get("NEVERCACHE_KEY", "")
ALLOWED_HOSTS = os.environ.get("ALLOWED_HOSTS", "")

DATABASES = {
    "default": {
        # Ends with "postgresql_psycopg2", "mysql", "sqlite3" or "oracle".
        "ENGINE": "django.db.backends.postgresql_psycopg2",
```



```

    # DB name or path to database file if using sqlite3.
    "NAME": os.environ.get("DATABASE_NAME", ""),
    # Not used with sqlite3.
    "USER": os.environ.get("DATABASE_USER", ""),
    # Not used with sqlite3.
    "PASSWORD": os.environ.get("DATABASE_PASSWORD", ""),
    # Set to empty string for localhost. Not used with sqlite3.
    "HOST": os.environ.get("DATABASE_HOST", ""),
    # Set to empty string for default. Not used with sqlite3.
    "PORT": os.environ.get("DATABASE_PORT", "")
}
}

SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTOCOL", "https")

CACHE_MIDDLEWARE_SECONDS = 60

CACHE_MIDDLEWARE_KEY_PREFIX = "mezzanine"

CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
        "LOCATION": os.environ.get("MEMCACHED_LOCATION", "memcached:11211"),
    }
}

SESSION_ENGINE = "django.contrib.sessions.backends.cache"

TWITTER_ACCESS_TOKEN_KEY = os.environ.get("TWITTER_ACCESS_TOKEN_KEY ", "")
TWITTER_ACCESS_TOKEN_SECRET = os.environ.get("TWITTER_ACCESS_TOKEN_SECRET ", "")
TWITTER_CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY ", "")
TWITTER_CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET ", "")
TWITTER_DEFAULT_QUERY = "from:ansiblebook"

```

Note how most of the settings in [Example 2-3](#) make reference to an environment variable by calling `os.environ.get`.

For most of the settings, we don't use a meaningful default value if the environment variable doesn't exist. There is one exception, the location of the memcached server:

```
"LOCATION": os.environ.get("MEMCACHED_LOCATION", "memcached:11211"),
```

I do this so that the default will handle the case where we use container linking. If I link the Memcached container with the name *memcached* at runtime, then Docker will automatically resolve the `memcached` hostname to the IP address of the Memcached container.

[Example 2-4](#) shows the Gunicorn configuration file. We could probably get away with hardcoding 8000 as the port, but instead I've allowed the user to override this by defining the `GUNICORN_PORT` environment variable.

*Example 2-4. gunicorn.conf.py*

```
from __future__ import unicode_literals
import multiprocessing
import os

bind = "0.0.0.0:{}".format(os.environ.get("GUNICORN_PORT", 8000))
workers = multiprocessing.cpu_count() * 2 + 1
loglevel = "error"
proc_name = "mezzanine"
```

The *setadmin.py* and *setsite.py* files are unchanged from the originals in Examples 6-17 and 6-18. We copy these into the container so that we can invoke them at deployment time. In our original playbook, we copied these files to the host at deploy time and executed them, but Docker doesn't yet support a simple way to copy files into a container at runtime, so instead we just copied them into the image at build-time.

## The Other Container Images

Our Mezzanine example uses some additional Docker images that we do not use Ansible to configure.

### Postgres

We need an image that runs the Postgres service. Fortunately, the Postgres project has an official image in the **Docker registry**. I'm going to use an official image, so there's no need for us to create our own. Specifically, I'm going to use the image that contains Postgres version 9.4, which is named `postgres:9.4`.

### Memcached

There's no official Memcached image, but the Dockerfile to build one is very simple, as shown in **Example 2-5**.

*Example 2-5. Dockerfile for Memcached*

```
FROM ubuntu:trusty
MAINTAINER lorin@ansiblebook.com

# Based on the Digital Ocean tutorial: http://bit.ly/1qJ8CXP

# Update the default application repository sources list
RUN apt-get update

# Install Memcached
RUN apt-get install -y memcached
```

```

# Port to expose (default: 11211)
EXPOSE 11211

# Default Memcached run command arguments
CMD ["-m", "128"]

# Set the user to run Memcached daemon
USER daemon

# Set the entrypoint to memcached binary
ENTRYPOINT memcached

```

## Nginx

There is an official **Dockerfile Nginx image** that we can use. We need to use our own configuration file for nginx so that it reverse proxies to the Mezzanine application. The official nginx image is configured so that we could put our custom *nginx.conf* file on the local filesystem of the host and mount it into the container. However, I prefer to create a self-contained Docker image that doesn't depend on configuration files that are outside of the container.

We can build a new image using the official image as a base and add our custom nginx configuration file into it. **Example 2-6** shows the Dockerfile and **Example 2-7** shows the custom nginx configuration file we use.

*Example 2-6. Dockerfile for custom nginx Docker image*

```

FROM nginx:1.7

RUN rm /etc/nginx/conf.d/default.conf \
    /etc/nginx/conf.d/example_ssl.conf
COPY nginx.conf /etc/nginx/conf.d/mezzanine.conf

```

*Example 2-7. nginx.conf for nginx Docker image*

```

upstream mezzanine {
    server mezzanine:8000;
}

server {

    listen 80;

    listen 443 ssl;

    client_max_body_size 10M;
    keepalive_timeout 15;
}

```

```

ssl_certificate      /certs/nginx.crt;
ssl_certificate_key  /certs/nginx.key;
ssl_session_cache    shared:SSL:10m;
ssl_session_timeout  10m;
ssl_ciphers (too long to show here);
ssl_prefer_server_ciphers on;

location / {
    proxy_redirect      off;
    proxy_set_header    Host                $host;
    proxy_set_header     X-Real-IP           $remote_addr;
    proxy_set_header     X-Forwarded-For     $proxy_add_x_forwarded_for;
    proxy_set_header     X-Forwarded-Protocol $scheme;
    proxy_pass            http://mezzanine;
}

location /static/ {
    root                /srv/project;
    access_log          off;
    log_not_found       off;
}

location /robots.txt {
    root                /srv/project/static;
    access_log          off;
    log_not_found       off;
}

location /favicon.ico {
    root                /srv/project/static/img;
    access_log          off;
    log_not_found       off;
}
}

```

Nginx doesn't natively support reading in configuration from environment variables, so we need to use a well-known path for the location of the static content (*/srv/project/static*). We specify the location of the Mezzanine service as *mezzanine:8000*; when we link the nginx container to the Mezzanine container; then Docker will ensure that the *mezzanine* hostname resolves to the Mezzanine container's IP address.

## Certs

The *certs* Docker image is a file that contains the TLS certificate used by nginx. In a real scenario, we'd use a certificate issued from a certificate authority. But for the purposes of demonstration, the Dockerfile for this image generates a self-signed certificate for *http://192.168.59.103.xip.io*, as shown in [Example 2-8](#).

*Example 2-8. Dockerfile for certs image*

```
FROM ubuntu:trusty
MAINTAINER lorin@ansiblebook.com

# Create self-signed cert for 192.168.59.103
RUN apt-get update
RUN apt-get install -y openssl

RUN mkdir /certs

WORKDIR /certs

RUN openssl req -new -x509 -nodes -out nginx.crt \
    -keyout nginx.key -subj '/CN=192.168.59.103.xip.io' -days 3650

VOLUME /certs
```

## Building the Images

I did not use Ansible itself to build the Docker images. Instead, I just built them on the command line. For example, to build the Mezzanine image, I wrote:

```
$ cd mezzanine
$ docker build -t lorin/mezzanine .
```

Ansible does contain a module for building Docker images, called `docker_image`. However, that module has been deprecated because building images isn't a good fit for a tool like Ansible. Image building is part of the build process of an application's lifecycle; building Docker images and pushing them up an image registry is the sort of thing that your continuous integration system should be doing, not your configuration management system.

# Deploying the Dockerized Application



We use the `docker` module for deploying the application. As of this writing, there are several known issues with the `docker` module that ships with Ansible.

- The `volumes_from` parameter does not work with recent versions of Docker.
- It does not support Boot2Docker, a commonly used tool for running Docker on OS X.
- It does not support the `wait` parameter that I use in some examples in this section.

There are proposed fixes for all of these issues awaiting review in the Ansible project. Hopefully by the time you read this, these issues all will have been fixed. There is also a pending pull request to support `detach=no`, which has the same behavior as `wait=yes` in the examples here. In the meantime, I have included a custom version of the `docker` module in the code sample repository that has fixes for these issues. The file is *ch13/playbooks/library/docker.py*.

**Example 2-10** shows the entire playbook that orchestrates the Docker containers in our Mezzanine deployment. The sensitive data is in a separate file, shown in **Example 2-11**. You can think of this as a development setup, because all of the services are running on the control machine.

Note that I'm running this on Mac OS X using Boot2Docker, so the Docker containers actually run inside of a virtual machine, rather than on localhost. This also means that I can invoke Docker without needing it to be root. If you're running this on Linux, you'll need to use `sudo` or run this as root for it to work.

Since this is a large playbook, let's break it down.

## Starting the Database Container

Here's how we start the container that runs the Postgres database.

```
- name: start the postgres container
  docker:
    image: postgres:9.4
    name: postgres
    publish_all_ports: True
    env:
      POSTGRES_USER: "{{ database_user }}"
      POSTGRES_PASSWORD: "{{ database_password }}"
```

Whenever you start a Docker container, you must specify the image. If you don't have the `postgres:9.4` image installed locally, then Docker will download it for you the first time it runs. We specify `publish_all_ports` so that Docker will open up the ports that this container is configured to expose; in this case, that's port 5432.

The container is configured by environment variables, so we pass the username and password that should have access to this service. The Postgres image will automatically create a database with the same name as the user.

## Retrieving the Database Container IP Address and Mapped Port

When we started up our Postgres container, we could have explicitly mapped the container's database port (5432) to a known port on the host. (We'll do this for the `nginx` container.) Since we didn't, Docker will select an arbitrary port on the host that maps to 5432 inside of the container.

Later on in the playbook, we're going to need to know what this port is, because we need to wait for the Postgres service to start before we bring up Mezzanine, and we're going to do that by checking to see if there's anything listening on that port.

We could configure the Mezzanine container to connect on the mapped port, but instead I decided to have the Mezzanine container connect to port 5432 on the Postgres container's IP address, which gives me an excuse to demonstrate how to retrieve a Docker container's IP address.

When the Docker module starts one or more containers, it sets information about the started container(s) as facts. This means that we don't need to use the `register` clause to capture the result of invoking this module; we just need to know the name of the fact that contains the information we're looking for.

The name of the fact with the information is `docker_containers`, which is a list of dictionaries that contains information about the container. It's the same output you'd see if you used the `docker inspect` command. [Example 2-9](#) shows an example of the value of the `docker_containers` fact after we start a postgres container.

*Example 2-9. docker\_containers fact after starting postgres container*

```
[
  {
    "AppArmorProfile": "",
    "Args": [
      "postgres"
    ],
    "Config": {
      "AttachStderr": false,
```

```

"AttachStdin": false,
"AttachStdout": false,
"Cmd": [
    "postgres"
],
"CpuShares": 0,
"Cpuset": "",
"Domainname": "",
"Entrypoint": [
    "/docker-entrypoint.sh"
],
"Env": [
    "POSTGRES_PASSWORD=password",
    "POSTGRES_USER=mezzanine",
    "PATH=/usr/lib/postgresql/9.4/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "LANG=en_US.utf8",
    "PG_MAJOR=9.4",
    "PG_VERSION=9.4.0-1.pgdg70+1",
    "PGDATA=/var/lib/postgresql/data"
],
"ExposedPorts": {
    "5432/tcp": {}
},
"Hostname": "71f40ec4b58c",
"Image": "postgres",
"MacAddress": "",
"Memory": 0,
"MemorySwap": 0,
"NetworkDisabled": false,
"OnBuild": null,
"OpenStdin": false,
"PortSpecs": null,
"StdinOnce": false,
"Tty": false,
"User": "",
"Volumes": {
    "/var/lib/postgresql/data": {}
},
"WorkingDir": ""
},
"Created": "2014-12-25T22:59:15.841107151Z",
"Driver": "aufs",
"ExecDriver": "native-0.2",
"HostConfig": {
    "Binds": null,
    "CapAdd": null,
    "CapDrop": null,
    "ContainerIDFile": "",
    "Devices": null,
    "Dns": null,
    "DnsSearch": null,

```



```

    "ExtraHosts": null,
    "IpcMode": "",
    "Links": null,
    "LxcConf": null,
    "NetworkMode": "",
    "PortBindings": {
        "5432/tcp": [
            {
                "HostIp": "0.0.0.0",
                "HostPort": ""
            }
        ]
    },
    "Privileged": false,
    "PublishAllPorts": false,
    "RestartPolicy": {
        "MaximumRetryCount": 0,
        "Name": ""
    },
    "SecurityOpt": null,
    "VolumesFrom": [
        "data-volume"
    ]
},
    "HostnamePath": "/mnt/sda1/var/lib/docker/containers/71f40ec4b58c3176030274afb025fbd3eb130fe79d4a6a69de473096f335e7eb/hostname",
    "HostsPath": "/mnt/sda1/var/lib/docker/containers/71f40ec4b58c3176030274afb025fbd3eb130fe79d4a6a69de473096f335e7eb/hosts",
    "Id": "71f40ec4b58c3176030274afb025fbd3eb130fe79d4a6a69de473096f335e7eb",
    "Image": "b58a816df10fb20c956d39724001d4f2fabdddec50e0d9099510f0eb579ec8a45",
    "MountLabel": "",
    "Name": "/high_lovelace",
    "NetworkSettings": {
        "Bridge": "docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.12",
        "IPPrefixLen": 16,
        "MacAddress": "02:42:ac:11:00:0c",
        "PortMapping": null,
        "Ports": {
            "5432/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "49153"
                }
            ]
        }
    },
    "Path": "/docker-entrypoint.sh",
    "ProcessLabel": "",
    "ResolvConfPath": "/mnt/sda1/var/lib/docker/containers/71f40ec4b58c3176030274afb025fbd3eb130fe79d4a6a69de473096f335e7eb/resolv.conf",

```

```

    "State": {
      "Error": "",
      "ExitCode": 0,
      "FinishedAt": "0001-01-01T00:00:00Z",
      "OOMKilled": false,
      "Paused": false,
      "Pid": 9625,
      "Restarting": false,
      "Running": true,
      "StartedAt": "2014-12-25T22:59:16.219732465Z"
    },
    "Volumes": {
      "/var/lib/postgresql/data": "/mnt/sda1/var/lib/docker/vfs/dir/4ccd3150c8d74b9b0feb56df928ac915599e12c3ab573cd4738a18fe3dc6f474"
    },
    "VolumesRW": {
      "/var/lib/postgresql/data": true
    }
  }
]

```

If you waded through this output, you can see that the IP address and mapped port are in the `NetworkSettings` part of the structure:

```

    "NetworkSettings": {
      "Bridge": "docker0",
      "Gateway": "172.17.42.1",
      "IPAddress": "172.17.0.12",
      "IPPrefixLen": 16,
      "MacAddress": "02:42:ac:11:00:0c",
      "PortMapping": null,
      "Ports": {
        "5432/tcp": [
          {
            "HostIp": "0.0.0.0",
            "HostPort": "49153"
          }
        ]
      }
    },
  },

```

Here's how we extract out the IP address (`172.17.0.12`) and the mapped port number (`49153`) and assign them to variables using the `set_fact` module: [source,yaml+jinja]

```

- name: capture database ip address and mapped port
  set_fact:
    database_host: "{{ docker_containers[0].NetworkSettings.IPAddress }}"
    mapped_database_port: "{{ docker_containers[0].NetworkSettings.Ports[
'5432/tcp'][0].HostPort }}"

```

# Waiting for the Database to Start Up

The [documentation for the official Postgres Docker image](#) contains the following caveat:

If there is no database when postgres starts in a container, then postgres will create the default database for you. While this is the expected behavior of postgres, this means that it will not accept incoming connections during that time. This may cause issues when using automation tools, such as fig, that start several containers simultaneously.

This is a great use case for the `wait_for` module, which will block playbook execution until the service accepts TCP connection requests:

```
- name: wait for database to come up
  wait_for: host={{ docker_host }} port={{ mapped_database_port }}
```

Note the use of the `docker_host` variable for specifying the host running Docker. Here's how this variable is defined up in the `vars` section. I've added a line break for clarity, but it should all be on one line.

```
docker_host: "{{ lookup('env', 'DOCKER_HOST') |
  regex_replace('^tcp://(.*):\\d+$', '\\\\1') | default('localhost', true) }}"
```

The issue is that the Docker host will depend on whether you're running on Linux, and therefore running Docker directly on your control machine, or whether you're running on Mac OS X, and are using Boot2Docker to run Docker inside of a virtual machine.

If you're running Docker locally, then `docker_host` should be set to `localhost`. If you're running Boot2Docker, then it should be set to the IP address of the virtual machine.

If you're running Boot2Docker, then you need to have an environment variable named `DOCKER_HOST` defined. Here's what mine looks like:

```
DOCKER_HOST=tcp://192.168.59.103:2375
```

I need to extract the `192.168.59.103` part of that, if `DOCKER_HOST` is defined. If it's not defined, then I want to default to `localhost`.

I used the `env` lookup plug-in to retrieve the value of the `DOCKER_HOST` environment variable:

```
lookup('env', 'DOCKER_HOST')
```

To extract the IP address, I used the `regex_replace` filter, which is a custom Jinja2 filter defined by Ansible that allows you to do regular expression (note the number of backslashes required):

```
regex_replace('tcp://(.*):\\d+$', '\\\\1')
```

Finally, I used the standard default Jinja2 filter to set a default value of localhost for the variable `docker_host` if the `DOCKER_HOST` environment variable wasn't defined. Because the `env` lookup returns an empty string, I needed to pass `true` as the second argument to the default filter to get it to work properly. See the [Jinja2 documentation](#) for more details:

```
default('localhost', true)
```

## Initializing the Database

To initialize the database, we need to run the Django `syncdb` and `migrate` commands. (In Django 1.7, you only need to run `migrate`, but Mezzanine defaults to Django 1.6).

We need to run the Mezzanine container for this, but instead of running Gunicorn, we want to pass it the appropriate `syncdb` and `migrate` commands, as well as run the `setsite.py` and `setadmin.py` scripts to set the site ID and the admin password.

```
- name: initialize database
  docker:
    image: lorin/mezzanine:latest
    command: python manage.py {{ item }} --noinput
    wait: yes
    env: "{{ mezzanine_env }}"
  with_items:
    - syncdb
    - migrate

- name: set the site id
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setsite.py
    env: "{{ setsite_env.update(mezzanine_env) }}"
    wait: yes

- name: set the admin password
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setadmin.py
    env: "{{ setadmin_env.update(mezzanine_env) }}"
    wait: yes
```

We use the `command` parameter to specify the `syncdb` and `migrate` commands.

We use the `wait` parameter so that the module will block until the process completes. Otherwise, we could have a race condition where the database setup has not completed yet when we start up Mezzanine.

Note the use of the `env` parameter to pass environment variables with the configuration information, including how to connect to the database service. I put all of the environment variables into a `mezzanine_env` variable that's defined like this:

```
mezzanine_env:
  SECRET_KEY: "{{ secret_key }}"
  NEVERCACHE_KEY: "{{ nevercache_key }}"
  ALLOWED_HOSTS: "*"
  DATABASE_NAME: "{{ database_name }}"
  DATABASE_USER: "{{ database_user }}"
  DATABASE_PASSWORD: "{{ database_password }}"
  DATABASE_HOST: "{{ database_host }}"
  DATABASE_PORT: "{{ database_port }}"
  GUNICORN_PORT: "{{ gunicorn_port }}"
```

When we set the site ID, we need to add the additional two environment variables, which I defined in a `setsite_env` variable:

```
setsite_env:
  PROJECT_DIR: "{{ project_dir }}"
  WEBSITE_DOMAIN: "{{ website_domain }}"
```

We need to combine the `mezzanine_env` and `setsite_env` dicts into a single dict and pass that to the `env` parameter.

Unfortunately, there's no simple way to combine two dicts in Ansible, but there's a workaround. Jinja2 has an `update` method that lets you merge one dictionary into another. The problem is that calling this doesn't return the merged dictionary; it just updates the state of the dictionary. Therefore, you need to call the `update` method, and then you need to evaluate the variable. The resulting `env` parameter looks like this:

```
env: "{{ setsite_env.update(mezzanine_env) }}"
```

## Starting the Memcached Container

Starting the Memcached container is straightforward. We don't even need to pass it environment variables since Memcached doesn't need any configuration information. We also don't need to publish any ports since only the Mezzanine container will connect to it via linking.

```
- name: start the memcached container
  docker:
    image: lorin/memcached:latest
    name: memcached
```

## Starting the Mezzanine Container

We link the Mezzanine container with the Memcached container and pass it configuration information via the environment.

We also run another container with the same image to run `cron`, since Mezzanine uses `cron` to update information from Twitter:

```

- name: start the mezzanine container
  docker:
    image: lorin/mezzanine:latest
    name: mezzanine
    env: "{{ mezzanine_env }}"
    links: memcached

- name: start the mezzanine cron job
  docker:
    image: lorin/mezzanine:latest
    name: mezzanine
    env: "{{ mezzanine_env }}"
    command: cron -f

```

## Starting the Certificate Container

We start the container that holds the TLS certificates. Recall that this container doesn't run a service, but we need to start it so that we can mount the volume into the nginx container.

```

- name: start the cert container
  docker:
    image: lorin/certs:latest
    name: certs

```

## Starting the Nginx Container

Finally, we start the Nginx container. This container needs to expose ports 80 and 443 to the world. It also needs to mount volumes for the static content and the TLS certificates. Finally, we link it to the Mezzanine container so that the Mezzanine hostname will resolve to the container that runs Mezzanine:

```

- name: run nginx
  docker:
    image: lorin/nginx-mezzanine:latest
    ports:
      - 80:80
      - 443:443
    name: nginx
    volumes_from:
      - mezzanine
      - certs
    links: mezzanine

```

And that's it! If you're running Docker locally on your Linux machine, you should now be able to access Mezzanine at <http://localhost> and <https://localhost>. If you're running Boot2Docker on OS X, you should be able to access it at the IP address of your Boot2Docker VM, which you can get by doing:

```
boot2docker ip
```

On my machine, it's at <http://192.168.59.103> and <https://192.168.59.103>, or you can use xip.io and access it at <https://192.168.59.103.xip.io.f1326.20>.

## The Entire Playbook

**Example 2-10** shows the entire playbook, and **Example 2-11** shows the contents of the `secrets.yml` file.

*Example 2-10. run-mezzanine.yml*

```
#!/usr/bin/env ansible-playbook
---
- name: run mezzanine from containers
  hosts: localhost
  vars_files:
    - secrets.yml
  vars:
    # The postgres container uses the same name for the database
    # and the user
    database_name: mezzanine
    database_user: mezzanine
    database_port: 5432
    unicorn_port: 8000
    docker_host: "{{ lookup('env', 'DOCKER_HOST') |
regex_replace('^tcp://(.*):\\d+$', '\\\\1') | default('localhost', true) }}"
    project_dir: /srv/project
    website_domain: "{{ docker_host }}.xip.io"
    mezzanine_env:
      SECRET_KEY: "{{ secret_key }}"
      NEVERCACHE_KEY: "{{ nevercache_key }}"
      ALLOWED_HOSTS: "*"
      DATABASE_NAME: "{{ database_name }}"
      DATABASE_USER: "{{ database_user }}"
      DATABASE_PASSWORD: "{{ database_password }}"
      DATABASE_HOST: "{{ database_host }}"
      DATABASE_PORT: "{{ database_port }}"
      GUNICORN_PORT: "{{ unicorn_port }}"
    setadmin_env:
      PROJECT_DIR: "{{ project_dir }}"
      ADMIN_PASSWORD: "{{ admin_password }}"
    setsite_env:
      PROJECT_DIR: "{{ project_dir }}"
      WEBSITE_DOMAIN: "{{ website_domain }}"

  tasks:
    - name: start the postgres container
      docker:
        image: postgres:9.4
        name: postgres
        publish_all_ports: True
```

```

    env:
      POSTGRES_USER: "{{ database_user }}"
      POSTGRES_PASSWORD: "{{ database_password }}"

- name: capture database ip address and mapped port
  set_fact:
    database_host: "{{ docker_containers[0].NetworkSettings.IPAddress }}"
    mapped_database_port: "{{ docker_containers[0].NetworkSettings.Ports[
'5432/tcp'][0].HostPort }}"

- name: wait for database to come up
  wait_for: host={{ docker_host }} port={{ mapped_database_port }}

- name: initialize database
  docker:
    image: lorin/mezzanine:latest
    command: python manage.py {{ item }} --noinput
    wait: True
    env: "{{ mezzanine_env }}"
  with_items:
    - syncdb
    - migrate
  register: django

- name: set the site id
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setsite.py
    env: "{{ setsite_env.update(mezzanine_env) }}"
    wait: yes

- name: set the admin password
  docker:
    image: lorin/mezzanine:latest
    command: /srv/scripts/setadmin.py
    env: "{{ setadmin_env.update(mezzanine_env) }}"
    wait: yes

- name: start the memcached container
  docker:
    image: lorin/memcached:latest
    name: memcached

- name: start the mezzanine container
  docker:
    image: lorin/mezzanine:latest
    name: mezzanine
    env: "{{ mezzanine_env }}"
    links: memcached

- name: start the mezzanine cron job
  docker:

```



```

    image: lorin/mezzanine:latest
    name: mezzanine
    env: "{{ mezzanine_env }}"
    command: cron -f

- name: start the cert container
  docker:
    image: lorin/certs:latest
    name: certs

- name: run nginx
  docker:
    image: lorin/nginx-mezzanine:latest
    ports:
      - 80:80
      - 443:443
    name: nginx
    volumes_from:
      - mezzanine
      - certs
    links: mezzanine

```

*Example 2-11. secrets.yml*

```

database_password: password
secret_key: randomsecretkey
nevercache_key: randomnevercachekey
admin_password: password

```

Because the Docker project is relatively young, much of the tooling is still in flux, and Docker deployment patterns are still evolving. Many of these emerging tools likely will have functionality that overlaps with Ansible when it comes to orchestrating containers.

Even if another tool emerges to dominate the world of Docker orchestration, I still believe Ansible will continue to be a useful tool for operators and developers.

## About the Author

---

**Lorin Hochstein** was born and raised in Montreal, Quebec, though you'd never guess he was a Canadian by his accent, other than his occasional tendency to say "close the light." He is a recovering academic, having spent two years on the tenure track as an assistant professor of computer science and engineering at the University of Nebraska-Lincoln, and another four years as a computer scientist at the University of Southern California's Information Sciences Institute. He earned his BEng. in Computer Engineering at McGill University, his MS in Electrical Engineering at Boston University, and his PhD in Computer Science at the University of Maryland, College Park. He is currently a Senior Software Engineer at SendGrid, where he works on new product development for SendGrid Labs.

## Colophon

---

The animal on the cover of *Ansible: Up and Running* is a Holstein Friesian (*Bos primigenius*), often shortened to Holstein in North America and Friesian in Europe. This breed of cattle originated in Europe in what is now the Netherlands, bred with the goal of obtaining animals that could exclusively eat grass—the area's most abundant resource—resulting in a high-producing, black-and-white dairy cow. Holstein-Friesians were introduced to the US from 1621 to 1664, but American breeders didn't become interested in the breed until the 1830s.

Holsteins are known for their large size, distinct black-and-white markings, and their high production of milk. The black and white coloring is a result of artificial selection by the breeders. Healthy calves weigh 90–100 pounds at birth; mature Holsteins can weigh up to 1280 pounds and stand at 58 inches tall. Heifers of this breed are typically bred by 13 to 15 months; their gestation period is nine and a half months.

This breed of cattle averages about 2022 gallons of milk per year; pedigree animals average 2146 gallons per year, and can produce up to 6898 gallons in a lifetime.

In September 2000, the Holstein became the center of controversy when one of its own, Hanoverhill Starbuck, was cloned from frozen fibroblast cells recovered one month before his death, birthing Starbuck II. The cloned calf was born 21 years and 5 months after the original Starbuck.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to [animals.oreilly.com](http://animals.oreilly.com).

The cover image is from Lydekker's *Royal Natural History*, Vol. 2. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.