



# Full Stack Python Guide to Deployments

by Matthew Makai

**The Full Stack Python Guide to Deployments** by Matthew Makai.

Copyright 2015 Matthew Makai. All rights reserved.

## Purpose

This book supplements the writing and material found on the [Full Stack Python](#)

website with completely new content. Full Stack Python (FSP) contains explanations of Python programming concepts, recommendations for how to learn and links to tutorials found around the web. However, FSP does not provide step-by-step instructions for performing an end-to-end Python web application deployment.

Over the past several years of writing FSP, I've received numerous requests from the programming community for more comprehensive details tutorials. This book is the result of those requests. My hope is that this guide helps folks as much as the FSP website.

## Revision History

2015-08-26: Typo and link fixes throughout the book.

2015-07-28: First edition. Initial release.

## Feedback and Updates

The plan is to update the book regularly based on reader feedback. Anyone who purchases the book will get access to the updates for free. To encourage early purchases and keep myself motivated to crank out new brand concept, over time I will raise the price slightly for people who haven't yet purchased the book. For example, if I knock out a new chapter with instructions on integrating unit tests into the continuous integration server, I may raise the price slightly from \$24 to \$26. Anyone

who's already purchased the book will not have to pay for a new copy or the difference between the new and old prices.

There are several scenarios that seem ripe for updates:

1. Incorporate updated software releases. For example, Ubuntu comes out with a new Long Term Support (LTS) version.
2. Write new content and chapters to expand the deployment in the book.
3. Fix or clarify issues with the walkthrough based on community feedback.

Let me know via email at matthew.makai@gmail.com whenever you run into confusing wording, inaccurate code snippets and general rough patches while going through the book. Thank you!

## About the Book's Cover

The cover photo is a picture I took while in Paris standing near the edge of the Le

Seine. The picture was then cropped and edited in Pixelmator to give it the sketch effect. A good portion of core content for the book was written in Paris which made it seem like an appropriate picture.

The picture for the cover was deliberately taken on the ground level, in contrast to the book edition of Full Stack Python which has a picture from above the clouds. This book is a hands-on, step-by-step guide for deployments that is complementary to the 20,000 foot deployments overview given by Full Stack Python.

## Thank you

To Char, for constant encouragement and telling me to "just release it already." To Luke and my parents, for inspiring me and listening to me say month after month that

it'd be released "any day now." To Twilio and the Developer Evangelism team, because I'd sure as hell never be able to write technical content at a level of excellence without what I've learned from you all since I started on the crew.

## Technical reviewers

Special thanks to my technical reviewers Andrew, Dylan and Kate for their feedback on numerous drafts of the book throughout many months of writing.

### Andrew Baker

Andrew Tork Baker is a software developer based in Washington, DC. He works at Twilio on their Developer Education team, devising new ways to teach people about programming. Though his current job exposes him to many different programming languages, Python and its community will always be his first choice. Andrew also organizes the [Django District](#) meetup in DC.

Andrew is the instructor for [O'Reilly's Introduction to Docker video](#), blogs at <http://www.andrewtorkbaker.com> and can be reached on [GitHub via atbaker](#) and [Twitter @andrewtorkbaker](#).

### Dylan Fox

Dylan Fox is a software developer at Cisco Systems, where he's part of a new Innovation team focused on building new collaboration technologies. He got into programming and Python while working on a startup he founded in college. Dylan can be reached on [GitHub via dylanbfox](#) and [Twitter @YouveGotFox](#).

## **Kate Heddleston**

Kate Heddleston is a web applications developer in the Bay Area who has been using Python and Django since graduation. She received her MS in CS at Stanford with an undergrad degree in communication. Kate enjoys using open-source tools to build web applications, and especially likes to build product features that interface with the user. She believes that open-source technologies are the foundation of our modern tech-driven world and that automation is one of the core values that technology offers us. Thus, open-source automation tools are some of Kate's favorite things in the world, just below puppies and just above shoe shopping. Kate can be reached on [GitHub via heddle317](#) and [Twitter @heddle317](#).

# Table of Contents

<b>Chapter 1: Introduction .....</b>	<b>8</b>
1.1 Who This Book Is For	10
1.2 Our Deployment	10
1.3 Deployment Automation	12
1.4 How to Use this Book	14
1.5 Services We'll Use	14
1.6 Our Example WSGI Application	15
1.7 Ready for Server Setup	17
<b>Chapter 2: Servers .....</b>	<b>18</b>
2.1 Hosting Options	20
2.2 What are Virtualized Servers?	21
2.3 Obtain your Virtual Server	22
2.4 Create Public and Private Keys	28
2.5 Boot the Server and Secure It	30
2.6 Upload the Public Key	35
2.7 Restart the SSH Service	36
2.8 Automating Server Setup	37
2.9 Next: Operating System	41
<b>Chapter 3: Operating Systems .....</b>	<b>42</b>
3.1 Ubuntu	44
3.2 Installing Ubuntu Packages	45
3.3 Enable the Firewall	46
3.4 Ansible	47
3.5 Automating Operating System Setup	47
3.6 Next: Web Server	51
<b>Chapter 4: Web Servers .....</b>	<b>53</b>
4.1 Web Servers in Our Deployment	55
4.2 Visualizing our Web Server's Purpose	56
4.3 Nginx	57
4.4 Installing Nginx	58
4.5 Domain Name Service Resolution	58
4.6 Configure Nginx Without HTTPS	62
4.7 Create an SSL Certificate	64
4.8 Install SSL Certificates	66
4.9 Configure Nginx with HTTPS	68

<b>4.10</b>	<b>Restarting Nginx</b>	<b>70</b>
4.11	Automating the Nginx Configuration	70
4.12	Next: Source Control	76
<b>Chapter</b>	<b>5: Source Control .....</b>	<b>78</b>
5.1	Hosted Source Control Services	81
5.2	Creating a Deploy Key	81
5.3	Authorizing Git Clone Access	82
5.4	Pulling Project Code to the Server	85
5.5	Automating Source Control	87
5.6	Next: Databases	88
<b>Chapter</b>	<b>6: Databases .....</b>	<b>90</b>
6.1	Databases in Our Deployment	92
6.2	PostgreSQL	93
6.3	Installing PostgreSQL	93
6.4	NoSQL Data Stores	95
6.5	Redis	96
6.6	Installing Redis	96
6.7	Testing the Redis Installation	96
6.8	Automating the PostgreSQL and Redis Installations	98
6.9	Next: Application Dependencies	100
<b>Chapter</b>	<b>7: Application Dependencies .....</b>	<b>103</b>
7.1	Application Dependencies in Our Deployment	105
7.2	Isolating Dependencies	106
7.3	Storing Dependencies in requirements.txt	106
7.4	Creating Our Virtualenv	107
7.5	Installing Dependencies	108
7.6	Syncing the Database Schema	110
7.7	Populating Our Application's Initial Data	110
7.8	Automating App Dependency Installation	111
7.9	Next: WSGI Server	115
<b>Chapter</b>	<b>8: WSGI Servers .....</b>	<b>117</b>
8.1	What is WSGI?	119
8.2	WSGI Server Implementations	120
8.3	Configuring Green Unicorn	121
8.4	Starting Gunicorn with Supervisor	122
8.5	Our App is Live!	123
8.6	Automating Gunicorn Setup	123
8.7	Next: Task Queue	125
<b>Chapter</b>	<b>9: Task Queues .....</b>	<b>127</b>
9.1	What Are Task Queues For?	129
9.2	Task Queue Implementations	130
9.3	Configuring Celery	131

9.4 Automating Celery Installation	132
9.5 Next: Continuous Integration	137
<b>Chapter 10: Continuous Integration .....</b>	<b>139</b>
10.1 How Will We Use Continuous Integration?	141
10.2 Setting Up Continuous Integration	142
10.3 Provisioning a New Server	143
10.4 Tweaking Our Ansible Automation	145
10.5 Jenkins Package Installation	148
10.6 Securing Jenkins with Authentication	150
10.7 Creating a CI Deploy Key	156
10.8 Creating Our Jenkins Build Job	160
10.9 Go Ahead, Push Some New Code	163
10.10 Next Steps	164
10.11 Other Open Source CI Projects	164
10.12 Hosted CI Services	165
<b>Chapter 11: What's Next? .....</b>	<b>167</b>
11.1 Deployment Enhancements	168
11.2 Improving Performance	169
11.3 Onward!	169
<b>Appendix A: Glossary .....</b>	<b>170</b>
<b>Appendix B: More Python Resources .....</b>	<b>177</b>
<b>Appendix C: Flask Application Code Tutorial .....</b>	<b>184</b>

Chapter 1

# Introduction

# Introduction

You've built the first version of your Python-powered web application. Now you want anyone around the world with an Internet connection to be able to use what you've created. However, before it's possible for others to access your application you need to configure a production server and deploy your code so it's running properly in that environment.

How do you go about handling the entire deployment? What about subsequent deployments when your code changes? Even if you've previously deployed a web application there are many moving pieces in the process that can be frustratingly difficult to handle.

That's where The Full Stack Python Guide to Deployments comes in. This book will guide you step-by-step through every task necessary to deploy a Python web application.

Each chapter in this book will teach you how to manually configure a part of the deployment pipeline and explain what you're accomplishing with that step. With the knowledge you learn from working through each step manually, we'll then use [Fabric](#) and [Ansible](#) to automate the deployment process. At the end of each chapter you'll not only understand what you're doing but you'll also have the steps automated for future deployments.

# Who This Book is For

If you're a beginner to intermediate Python developer who's learned the basics of building a web application and wants to learn more about deploying web apps, then this is exactly the book for you.

There are many deployment tutorials on the web, many of which are linked to on [Full Stack Python](#) and in the chapters of this book. However, you'd need to piece together many of those tutorials and read between the lines to get a complete end-to-end deployment successfully completed. With this book, you won't need to worry about guessing what to do next. We will walk through every step and explain it so you know what you're doing and why it needs to be done.

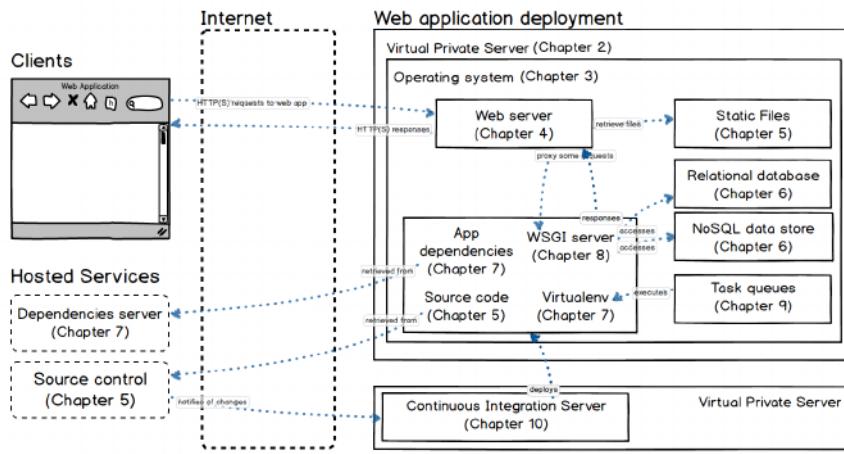
If you've already deployed numerous Python web applications and are managing several Python apps in production, this isn't the book for you because we are starting with the assumption that you don't have any pre-existing knowledge about deployments.

# Our Deployment

Throughout this book we'll work through setting up the infrastructure to run a production version of a Python web application. All code will be deployed on a single virtual private server.

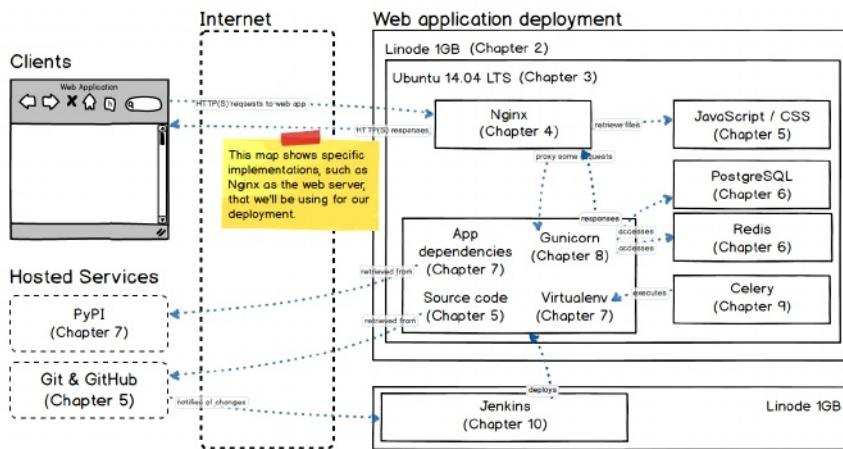
It's okay if some of the technical terms, such as production server or WSGI, are confusing to you! You'll learn their definitions and how the pieces fit together as we go along. There's also a technical terms appendix for reference that you can find near the end of the book.

The picture below this paragraph is a deployment concepts map of the book's content. Each chapter in this book contains this map with a highlight on the concept and software we will configure in that chapter.



The above map shows concepts such as web servers and web frameworks. These concepts are abstractions, not specific implementations of Python software projects. For example, the abstract concept of a web server is implemented by [Nginx](#) (pronounced "Engine-X") server, which we'll install and configure in chapter 4.

The below map has the same structure as the above map but replaces the concepts with their implementations for our deployment pipeline.



Again, don't worry if you're unfamiliar with the concepts or implementations shown in the above maps. Each chapter will introduce a concept, explain how to set up the implementation manually, automate the steps and give additional resources to continue learning more advanced topics on the subject.

## Deployment Automation

This book teaches you how to automate your application deployments even if you've never done systems administration work before. Once you understand how deployments work and have automation in place, you'll be able to quickly iterate on your code in your development environment then immediately get working code out to your production server.

Automation is critical for keeping running web applications up to date with the most recently developed code. Once your application is live on the production server, users will give you feedback, request changes and discover bugs. The faster you can fix those issues and add enhancements the better chance your application will succeed. Automated deployments provide the speed and reliability to keep the code running in production up to date even if you make many changes each day.

Every step in the deployment is automated as we work our way through the chapters. There is an [open source repository on GitHub](#) with the automation code, as well as Git tags for the incremental steps performed in each chapter. The following links take you to each chapter's corresponding tag on GitHub:

- [01-introduction](#): just the README and a stub directory for SSH keys
- [02-servers](#): Fabric `fabfile.py` in the `prod` subdirectory
- [03-operating systems](#): start of the Ansible playbook
- [04-web-servers](#): builds Nginx configuration into the Ansible playbook
- [05-source-control](#): adds Git repository cloning to the playbook
- [06-databases](#): sets up PostgreSQL and Redis
- [07-application-dependencies](#): installs Python packages into a virtualenv, establishes environment variables and syncs the app to the database
- [08-wsgi-servers](#): configures Supervisor to run the app with Green Unicorn
- [09-task-queues](#): adds to the Supervisor configuration for Celery and Redis
- [10-continuous-integration](#): modifies the Ansible playbook so it does not prompt for SUDO password for CI server

Tags may be added in the future when new chapters are added to this book.

# How to Use this Book

If you've never deployed a Python web application before my recommendation is to go through the book twice. First, go through the manual steps and take your time understanding the individual components, such as WSGI servers, as well as the overall picture of how the implementations fit together once they're deployed. Once you understand the manual deployment process then set up a second, separate server and work through each chapter's deployment automation steps.

Learning how to automate deployments with Ansible will be most useful for folks who have already manually deployed web applications and have a grasp on basic systems administration. If you're coming to this book with prior deployment experience then skim through the manual steps in each chapter before moving on to the automated deployment instructions.

# Services We'll Use

In this book we'll use several services to get our environment up and running. You don't need to set these accounts up right now as we'll walk through them in the chapters where they're necessary. If you have accounts for these services already then you're one step ahead. The services are

- [Linode](#): hosts our virtual private production server
- [GitHub](#): handles source control in the deployment
- [Twilio](#)(optional): automates deployment alerts

GitHub and Twilio can be used with the free trial account for our purposes.

We'll also need several free open source projects to handle the deployment, including

- [Fabric](#): provides a convenient Python code wrapper for SSH commands
- [Ansible](#): an easy to use but powerful configuration management automation tool
- [Nginx](#): the second most common web server currently deployed that is very popular with the Python community
- [Green Unicorn](#): a Web Server Gateway Interface (WSGI) server that'll execute our Python code so we can run our web application

You'll also need a local Linux or Mac OS X environment. If you're on Windows you should stand up [VirtualBox](#) on your machine to run Ubuntu Linux 14.04 LTS. Here's [a handy tutorial for installing VirtualBox on Windows](#) if you need to do that now.

## Our Example WSGI Application

Throughout the book we're going to use an open source project as the example code to deploy. Our web application uses the [Flask web framework](#) and follows the Python [Web Server Gateway Interface](#) (WSGI) standard defined by the Python community in [PEP 3333](#) for deploying web applications to web application servers. PEP 3333 specifies that [WSGI frameworks](#) such as [Django](#), [Flask](#) and [Pyramid](#) can be interchangeably deployed to [WSGI servers](#) such as [Gunicorn](#), [mod\\_wsgi](#) and [uWSGI](#).

Our example WSGI application in this book is a Flask project which serves up [Reveal.js](#) presentations that allow live audience voting by text messages. The votes are calculated in the Flask application and immediately displayed in the presentation via a [WebSocket](#) connection. The app is called [Choose Your Own Deployment Adventure](#)

[Presentations](#). The code was used at [DjangoCon US 2014](#) for a talk named "[Choose Your Own Django Deployment Adventure](#)" given by [Kate Heddleston](#) and myself. The following screenshot shows what the application looks like with the default presentation styling.



A benefit of using the Choose Your Own Deployment Adventure Presentations code as our example is that there is a detailed walkthrough for building the application ([part 1](#), [part 2](#), [part 3](#) and [part 4](#)) on the [Twilio blog](#). The full four-part tutorial is also included as Appendix C in this book if you prefer to work through it that way. The code is well documented and released under the MIT license.

## Ready for Server Setup

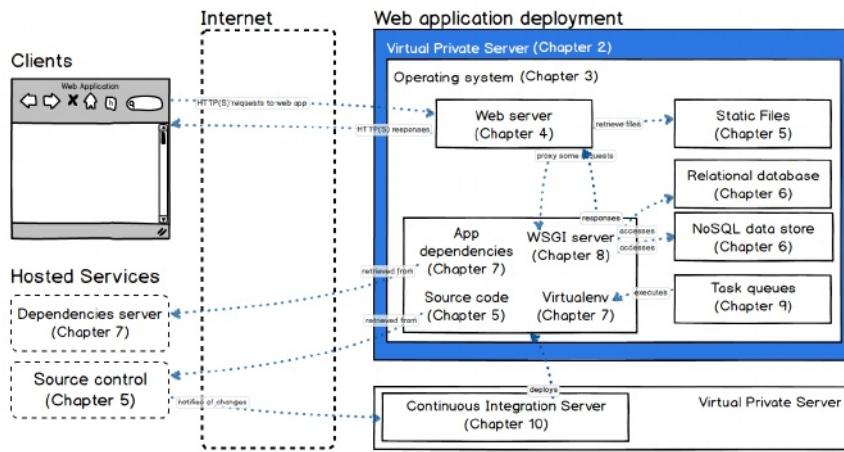
Let's get started with our web application deployment by obtaining a production server from Linode.

Chapter 2

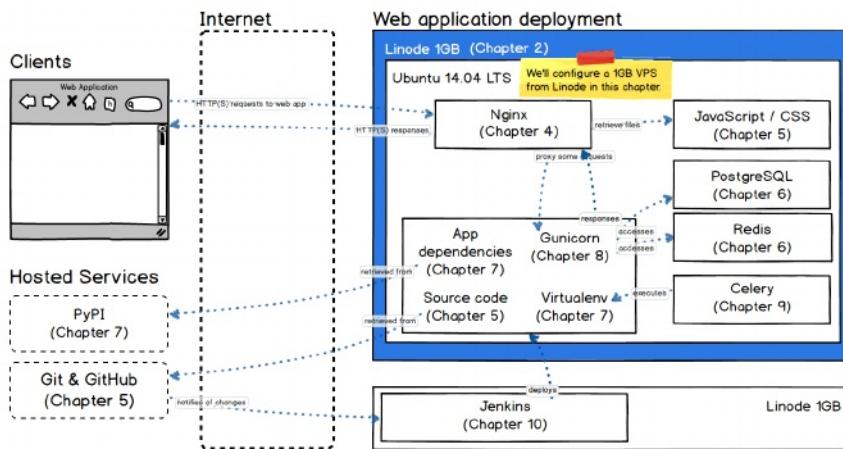
# Servers

# Servers

Your web application must run somewhere other than your development environment on your desktop or laptop. That location is a separate server (or cluster of servers) known as a “production environment”. Throughout this book, we’ll use a single server for our application’s production environment deployment.



In this chapter we'll take a look at several hosting options, obtain a virtual private server to use as our production environment, boot up the server and secure it against unauthorized access attempts.



As shown in the above diagram, [Linode](#)'s cloud offering will provide us with a virtual private server for our production environment in this deployment.

## Hosting Options

Let's consider our hosting options before diving into the deployment on the Linode virtual private server. There are four general options for deploying and hosting a web application:

- *"Bare metal/" servers*: physical hardware; vendor examples: Dell, Lenovo
- *Virtual/private servers*: shared rented portions of servers, examples: Linode, Digital Ocean
- *Infrastructure-as-a-service (IaaS)*: virtualized abstractions of resources, examples: Amazon Web Services & Microsoft Azure

- *Platform-as-a-service (PaaS)* abstracted execution environments such as Heroku, Python Anywhere or Amazon Elastic Beanstalk

The first three options are similar. The deployer needs to provision one or more servers with a Linux distribution. System packages, a web server, WSGI server, database and the Python environment are then installed. The application code can be pulled from a source controlled repository and configured in the environment.

We'll deploy our web application on a virtual private server (VPS). There are plenty of VPS hosting options. I've used Linode for over five years now. They're a stable company with solid support when issues occur, which is rarely. You typically get what you pay for in the VPS hosting industry.

If you're considering a different provider make sure to check the resources section at the end of this chapter for evaluating VPS alternatives.

## What are Virtualized Servers?

Virtual private servers (VPSs) are virtualized slices of hardware run on top of a physical server. Virtualization software such as [Xen](#) and [VMWare](#) allow a providers' customers to use fractions of a full server that appear as their own independent instances. For example, a server with an 8-core processor and 16 gigabytes of memory can be roughly virtualized into 8 pieces with the equivalent of 1-core and 2 gigabytes of memory.

The primary disadvantage of virtualized servers is that there is resource overhead in the virtualization process. But for our web application deployment, a single well-configured virtual private server provides more than enough performance and represents a huge cost savings over purchasing dedicated hardware.

Let's obtain a Linode account, provision a server and get our deployment started.

## Obtain your Virtual Server

These steps sign you up for a Linode account and guide you through provisioning a virtual private server for \$10/month which we'll configure throughout the rest of the book.

A note before we get started. Throughout this book the instructions will follow a standard format. Each step will explain what to do along with some context, such as a screenshot or snippet of code you'll need to type in. Within the code snippets there are lines prefixed with `#` that are comments. You don't have to type the comments in, they just provide additional context for what the commands are specifically doing and why the steps are necessary.

With that note out of the way, let's provision a Linux server so we can begin the deployment.

1. Point your web browser to [Linode.com](https://Linode.com). Their landing page will look something like the following image.



2. [Sign up](#) for an account.

A screenshot of the Linode account creation form. The title is "Create a free account". It contains three input fields: "Email", "Username", and "Password". Below these fields is a "Create Account" button. At the bottom of the form, there is a note about cookies and links for forgot username, reset password, and the website address.

3. You'll be sent an email for account confirmation. Fill out the appropriate information and add initial credit to your account. If you want to enter a referral code, mine is **bfeecaf55a83cd3dd224a5f2a3a001fdf95d4c3** to your account

will go for a quick review to ensure you're not a malicious spam bot and then the account will be fully activated.

4. Once your account is activated refresh thepage. You'll be given a page to add a Linode instance.
5. Select the 1024 option, month-to-month billing and the data center location of your choice. I chose Newark, NJ because I grew up in northern NJ and otherwise the location isn't important for my deployment. If your most of your users are located in a specific country or region you'll want to select the data center location closest to them.

The screenshot shows the Linode Manager interface for adding a new Linode. At the top, there's a navigation bar with links for Linodes, NodeBalancers, Longview, DNS Manager, Account, Support, Documentation, and Community. The user is logged in as 'mattdemo'. Below the navigation, a breadcrumb trail shows 'Linodes > Add a Linode'. A section titled 'Select your plan' displays several plan options in boxes:

- Linode 1024**: 24GB DISK, 2TB XFER, .015/hr to \$10/mo
- Linode 2048**: 48GB DISK, 3TB XFER, .03/hr to \$20/mo
- Linode 4096**: 96GB DISK, 4TB XFER, .06/hr to \$40/mo
- Linode 8192**: 192GB DISK, 8TB XFER, .12/hr to \$80/mo
- Linode 16384**: 384GB DISK, 16TB XFER, .24/hr to \$160/mo
- Linode 32768**: 768GB DISK, 20TB XFER, .48/hr to \$320/mo
- Linode 49152**: 1152GB DISK, 20TB XFER, .72/hr to \$480/mo
- Linode 65536**: 1536GB DISK, 20TB XFER, .96/hr to \$640/mo
- Linode 98304**: 1920GB DISK, 20TB XFER, 1.44/hr to \$960/mo

Below the plans, a 'Location' dropdown is set to 'Newark, NJ'. At the bottom right is a large blue button labeled 'Add this Linode!'

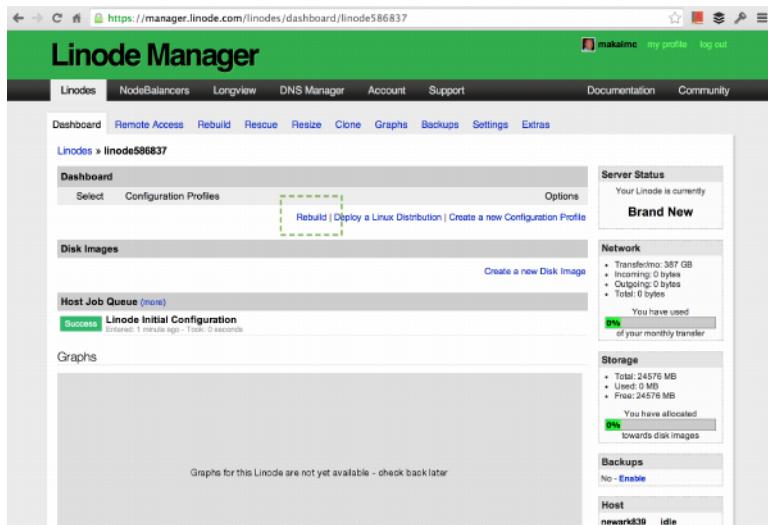
6. Click "Add this Linode!" and a dashboard will appear where we'll see the Linode is being created.

The screenshot shows the Linode Manager dashboard. At the top, there's a navigation bar with links for Linodes, NodeBalancers, Longview, DNS Manager, Account, Support, Documentation, and Community. A user profile icon for 'mattdemo' is also present. Below the navigation is a table titled 'Linodes'. The table has columns for Linode, Status, Plan, IP, Location, Backups, and Options. One row is visible, showing 'Linode586867' with 'Being Created' status, 'Linode 1024' plan, '50.116.81.9' IP, 'Newark, NJ, USA' location, 'No - Enable' backups, and 'Dashboard | Remove' options. Below the table is a section titled 'This Month's Network Transfer Pool' showing '100% Remaining' with '0GB Used, 368GB Remaining, 368GB Quota'. A note states: 'Your transfer is prorated and will reset next month'.

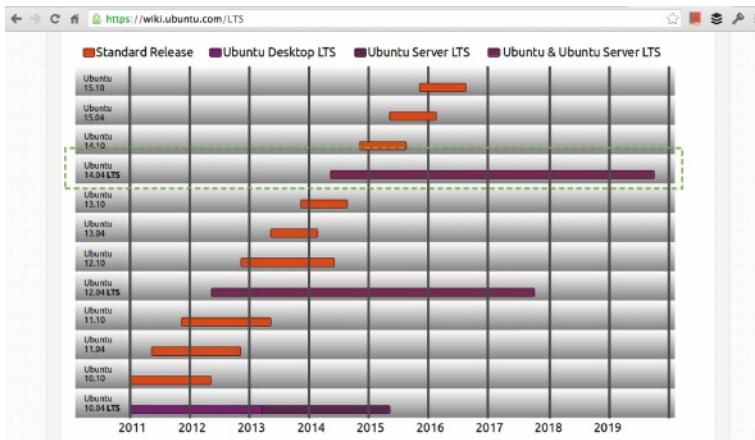
7. Refresh the page and look for the status to change to "Brand New." Write down or remember the IP address as we'll need it later to SSH into the server, then click on the name of the Linode. A page will appear to show more information about your new virtual private server.

This screenshot is identical to the one above, showing the Linode Manager dashboard with a single Linode entry. The difference is that the 'Status' column for 'Linode586867' now displays 'Brand New' instead of 'Being Created'. The rest of the interface remains the same, including the transfer pool summary at the bottom.

8. Click the "Rebuild" link.



9. We're going with Ubuntu 14.04 LTS for our deployment. This version of Ubuntu is the latest LTS (Long Term Support) release and has a 5 year support lifecycle. Therefore this version will receive security updates until April 2019 as shown on the [Ubuntu wiki page for LTS releases](#).



10. Select Ubuntu 14.04 LTS and enter a password. Make sure you type the password in carefully and remember it! We'll need the password again in a few minutes to log in as our root user. The "Deployment Disk Size" and "Swap Disk" can be left as they are.

The screenshot shows the Linode Manager interface for rebuilding a specific node. The URL is <https://manager.linode.com/linodes/rebuild/linode586867>. The page title is "Linode Manager". The navigation bar includes "Linodes", "NodeBalancers", "Longview", "DNS Manager", "Account", "Support", "Documentation", and "Community". The main menu has options like "Dashboard", "Remote Access", "Rebuild" (which is selected), "Rescue", "Resize", "Clone", "Graphs", "Backups", and "Settings". The sub-menu for "Rebuild" shows the path "Linodes » linode586867 » Rebuild". The form titled "Deploy a Linux Distribution" has the following fields:

- Distribution:** Ubuntu 14.04 LTS (selected)
- Deployment Disk Size:** 24320 MB (750 MB min, 24320 MB max)
- Swap Disk:** 256 MB (750 MB min, 24320 MB max)
- Root Password:** (Field highlighted with a green dashed box, with a callout note: "Fill this in and remember the password. You'll need it shortly.")
- Rebuild** button

11. Linode will send us back to our server's dashboard when the build process begins. We'll see the progress bars start to increase and in a couple of minutes it'll be ready to boot up.

However, don't start up the server just yet. We need to create a public-private key pair that will be used to harden our new server against unauthorized login attempts.

## Create Public and Private Keys

Before we boot up our newly-provisioned server we need to create a public-private key pair on our local machine. We'll then upload the locally-created key pair to our remote server.

Once we are using the key pair we can disable password logins for increased security. In addition, we won't have to type in a password each time we want to access the server via SSH, which is super convenient for quick logins.

If you already have an SSH key pair you want to use then feel free to skip on to the next section. However, if your existing key pair has a passphrase on it you'll want to follow these steps to create a new key pair. This book uses a passphrase-less key pair to perform the automated deployments as well as minimize the typing during our manual deployment process.

1. Open a new Mac OS X or Linux terminal window on your local computer (not the server you created on Linode). Again, if you are on Windows you'll need to [set up VirtualBox with a Linux instance](#) to follow these commands.
2. Create a local directory with the following commands to store our key pair. Remember that there is also a [companion GitHub repository](#) with all

this code and tags for each chapter in case you do not want to type everything in yourself. However, even if you use the code in the repository you'll have to create your own public-private key pair with these steps since the key pair cannot be shared.

```
# the mkdir command with the -p argument will recursively
# create the directory and subdirectory
mkdir -p fsp-deployment-guide/ssh_keys

# 'cd' moves us into ssh_keys directory in our local shell
cd fsp-deployment-guide/ssh_keys
```

3. Throughout this book we will refer to the `fsp-deployment-guide` directory as the *base directory* for our deployment files.
4. Execute the following `ssh-keygen` command to create the public-private key pair.

```
# we're running ssh-keygen command to generate the key pair
# -b argument specifies the number of bits, 2048 in our case
# along with -t argument to specify the RSA algorithm
ssh-keygen -t rsa -b 2048
```

5. When prompted for a file in which to save the keys do not use the default. Instead, enter the following directory and file into the prompt.  

```
# we are saving the private key in our local ssh_keys directory
./prod_key
```
6. Press enter twice when prompted for a passphrase. We will not use a passphrase for the deployer user's keys.
7. The key pair will be generated by `ssh-keygen`. Now you have two new files: a private key `prod_key` and a public key `prod_key.pub`. The private key should never be shared with anyone as it will allow access to your server once it's

setup. The public key can be shared with anyone and does not need to be kept private.

8. Execute the following command to create one more new file with the newly created public key as an authorized key.

```
# the authorized_keys file will allow our server to grant SSH
# access to a connection using the private key
cp prod_key.pub authorized_keys
```

You now have the required public and private key files as well as a `authorized_keys` file. Keep backups of these files in a safe place since once the server is locked down you'll need them to log in.

With our new key files we can now fire up the server and set up our security configuration.

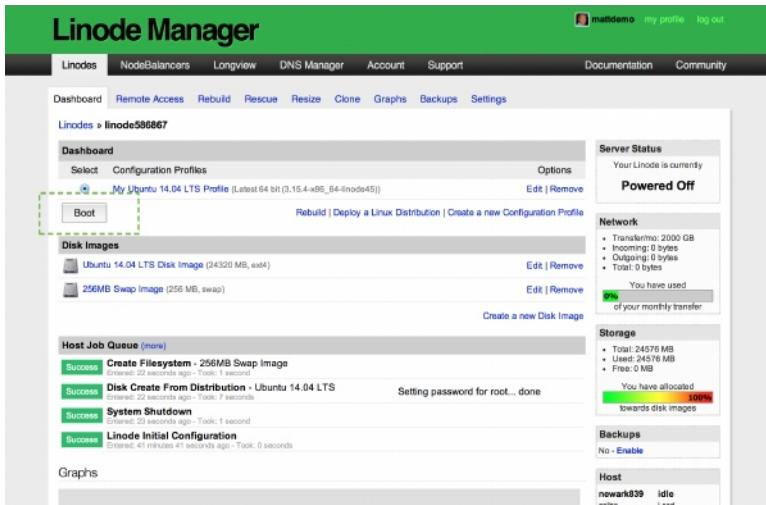
## Boot the Server and Secure It

When we boot up the server we need to ensure it is locked down against unauthorized access attempts. There are scores of malicious bots that search for servers across the IPv4 address range. The Internet can be a dark place for unsecured servers.

Fortunately, with a few steps we can mitigate the most glaring security weaknesses of a fresh server.

When you're ready to begin the initial lock down steps we can start our machine up.

1. Click the "Boot" button and the Ubuntu 14.04 LTS boot process will get started. Booting should take less than a minute. Bring up your local command line as we'll need it to connect to the remote machine.



2. SSH into your server with `ssh root@{ip.address.here}`, where `{ip.address.here}` is your server's IP address, which can be found on the Linode dashboard. For example, if your new Linode's IP address is 66.175.209.129, you'll

enter `ssh root@66.175.209.129`

3. You'll likely receive a prompt like the following warning. This prompt is stating that you've never connected to this server before, so are you sure that this host's signature matches the server you're intending to connect to? Enter yes then enter the root password you created during the earlier Linode server provisioning step.

```
The authenticity of host '66.175.209.192 (66.175.209.192)' can't be established.
RSA key fingerprint is 51:3c:ba:bc:c3:83:1a:36:b1:2d:e3:f6:6d:f0:11:56.
Are you sure you want to continue connecting (yes/no)? yes
```

4. You will see a "Welcome to Ubuntu 14.04.3 LTS" message followed by a prompt. Now we can enter commands on the remote machine to get the server secured and setup.

5. Ensure the system packages cache is up to date.

```
# this command tells apt, Ubuntu's package manager, to make
# sure it has the latest list of package versions to install
apt-get update
```

6. Next upgrade all out of date packages.

```
# apt will upgrade older packages to the newest versions
# available in the packager manager
apt-get upgrade
```

7. Let's install `fail2ban` with its sensible default settings.

```
# fail2ban is a security package that logs unsuccessful logins
# and prevents brute force login attempts
apt-get install fail2ban
```

8. Next we're going to modify the SSH configuration so the root user

account cannot be directly logged into. Only someone using the exact private key we created earlier can connect to this server.

9. Open the `/etc/ssh/sshd_config` file in the text editor of your choice. For example using `vim /etc/ssh/sshd_config`. That will open the file in Vim, but you can also use the `nano` or `emacs` commands instead. You should see the following lines of the file in your editor.

```
# Package generated configuration file
# See the sshd_config(5) manpage for details

# What ports, IPs and protocols we listen for
Port 22

# Use these options to restrict which interfaces/protocols sshd will bind to
```

```

#ListenAddress ::

#ListenAddress 0.0.0.0
Protocol 2
# HostKeys for protocol version 2
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
HostKey /etc/ssh/ssh_host_ecdsa_key
HostKey /etc/ssh/ssh_host_ed25519_key
#Privilege Separation is turned on for security
UsePrivilegeSeparation yes

# Lifetime and size of ephemeral version 1 server key
KeyRegenerationInterval 3600
ServerKeyBits 1024

# Logging
SyslogFacility AUTH
LogLevel INFO

# Authentication:
LoginGraceTime 120
PermitRootLogin yes
StrictModes yes

RSAAuthentication yes

PubkeyAuthentication yes
#AuthorizedKeysFile %h/.ssh/authorized_keys

# Don't read the user's ~/.rhosts and ~/.shosts files
IgnoreRhosts yes
# For this to work you will also need host keys in /etc/ssh_known_hosts
RhostsRSAAuthentication no
# similar for protocol version 2
HostbasedAuthentication no
# Uncomment if you don't trust ~/.ssh/known_hosts for RhostsRSAAuthentication
#IgnoreUserKnownHosts yes

# To enable empty passwords, change to yes (NOT RECOMMENDED)
PermitEmptyPasswords no

# Change to yes to enable challenge-response passwords (beware issues with
# some PAM modules and threads)

```

```
ChallengeResponseAuthentication no

# Change to no to disable tunneled clear text passwords
#PasswordAuthentication yes
```

10. Edit `#PasswordAuthentication yes` to the following.

```
# passwords are not necessary for logging in with the private key
PasswordAuthentication no
```

11. Modify `UsePAM yes` to the following.

```
# because we're using a keypair password authentication should be
# disabled for the server
UsePAM no
```

12. Change `#PermitRootLogin yes` to the following.

```
# lock down root user account and only log in with the account
# we are about to create
PermitRootLogin no
```

13. Save the file.

14. Next we need to create a non-root group for the new non-root user we'll create in a few steps. Execute the following commands. You can change "deployers" to another group name if you want but you have to be consistent in future steps.

```
# create the new deployers group
/usr/sbin/groupadd deployers
# back up the sudoers file
mv /etc/sudoers /etc/sudoers-backup
# modify the sudo list so the deployers group has sudo privileges
(cat /etc/sudoers-backup ; echo "%deployers ALL=(ALL) ALL") > /etc/sudoers
# ensure the appropriate permissions are on the sudoers file
chmod 0440 /etc/sudoers
```

15. Now add a non-root user with the non-root group we just created.

Replace the name `Matt Makai` with your name (but leave the quotes around

the name). You will be prompted to enter the same new password twice for this account.

```
# create the new user. be sure to use your own name here
/usr/sbin/useradd -c "Matt Makai" -m -g deployers deployer
# set up a password for the new user. you'll need the
# password to run sudo commands
/usr/bin/passwd deployer
```

16. Create a directory for the new deployer user to store our public and private keys.

```
# add the deployer user to the deployers group
/usr/sbin/usermod -a -G deployers deployer
# create a directory for the deployer's public key and
# authorized_keys file
mkdir /home/deployer/.ssh
# change the owner and group of the .ssh directory to deployer
# and deployers, respectively
chown -R deployer /home/deployer/.ssh
chgrp -R deployers /home/deployer/.ssh
```

Don't log out of root just yet. We need to upload the public and private keys for the deployer user so we can log in with that account.

## Upload the Public Key

In order to authenticate with our new deployer user, we need to upload the `prod.pub` and `authorized_keys` files.

1. Keep the root SSH connection open. We'll need that in a minute.
2. Go back to the command prompt on your local machine where the SSH keys we created earlier in this chapter are stored.

3. Upload the public SSH key and `authorized_keys` files to our server for the new deployer user. Again, make sure to replace the IP address with your server's IP address.

```
# scp is the 'secure copy' command and will transfer the
# public key along with the authorized_keys file to the server
scp prod_key.pub authorized_keys deployer@{your.server.ip.address}:~/ssh
```

4. When prompted by the `scp` command enter the deployer's password that was created in the previous section.

## Restart the SSH Service

1. Ensure that we can SSH into the new deployer user after we reload our SSH server configuration on the remote server. In the terminal where root is logged in, run this command. **Do not yet log off of root!**

```
# applies new SSH configuration settings so we can test them
service ssh reload
```

2. Now try to SSH in with the new deployer user in the other terminal window. Again replace the IP address in this command with the IP address of your server.

```
# connecting to our new user instead of the root user
ssh -i ./prod_key deployer@{your.server.ip.address}
```

3. If you get a command prompt on the server, success! We can now close out our root connection! You won't be able to log directly into root any

longer, only into the deployer user. That's a really good thing because automated malicious bots often try to log into root with their attacks.

## Automating Server Setup

Clearly we don't want to have to manually type in all these steps every time we create a new VPS instance. Fortunately the Fabric library makes these instructions easy to automate. Let's take a look at a short script that runs every step we just performed above. Remember, there is a [GitHub repo](#) with the automated code in case you want to copy and paste the text or clone the repository.

1. On your local machine, create a new directory named `prod` under your project. `prod` will store files for handling deployments to our production server for our application.
2. Within `prod` create a new file named `fabfile.py` with the following contents.

```
from os import environ
from fabric.api import *
from fabric.context_managers import cd
from fabric.contrib.files import sed

"""
Fabric file to upload public/private keys to remote servers
and set up non-root users. Also prevents SSH-ing in with the
root user. Fill in the following blank fields then run this
Fabric script with "fab bootstrap".
"""

# run the bootstrap process as root before it is locked down
env.user = 'root'

# the remote server's root password
```

```

env.password = ''

# all IP address or hostnames of the servers you want to put
# your SSH keys and authorized_host files on, ex: 192.168.1.1
env.hosts = ['192.168.1.1']

# your full name for the new non-root user
env.new_user_full_name = 'Matt Makai' # ex: Matt Makai

# username for the new non-root user to be created
env.new_user = 'deployer' # ex: deployer

# group name for the new non-root user to be created
env.new_user_grp = 'deployers' # ex: deployers

# local filesystem directory where your prod_key.pub and
# authorized_keys files are located (they will be scp'd
# to target hosts) don't include a trailing slash
# note: the tilde resolves to your home directory
env.ssh_key_dir = '~/fsp-deployment-guide/ssh_keys'

"""

The following functions should not need to be modified to
complete the bootstrap process.
"""

def bootstrap():
    local('ssh-keygen -R %s' % env.host_string)
    sed('/etc/ssh/sshd_config', '^UsePAM yes', 'UsePAM no')
    sed('/etc/ssh/sshd_config', '^PermitRootLogin yes', 'PermitRootLogin no')
    sed('/etc/ssh/sshd_config', '^#PasswordAuthentication yes',
        'PasswordAuthentication no')
    _create_privileged_group()
    _create_privileged_user()
    _upload_keys(env.new_user)
    run('service ssh reload')

def _create_privileged_group():
    run('/usr/sbin/groupadd ' + env.new_user_grp)
    run('mv /etc/sudoers /etc/sudoers-backup')
    run('(cat /etc/sudoers-backup ; echo "%" + env.new_user_grp \
        + ' ALL=(ALL) ALL") > /etc/sudoers')
    run('chmod 440 /etc/sudoers')

```

```

def _create_privileged_user():
    run('/usr/sbin/useradd -c "%s" -m -g %s %s' % \
        (env.new_user_full_name, env.new_user_grp, env.new_user))
    run('/usr/bin/passwd %s' % env.new_user)
    run('/usr/sbin/usermod -a -G ' + env.new_user_grp + ' ' + \
        env.new_user)
    run('mkdir /home/%s/.ssh' % env.new_user)
    run('chown -R %s /home/%s/.ssh' % (env.new_user,
        env.new_user))
    run('chgrp -R %s /home/%s/.ssh' % (env.new_user_grp,
        env.new_user))

def _upload_keys(username):
    local('scp ' + env.ssh_key_dir + \
        '/prod_key.pub ' + env.ssh_key_dir + \
        '/authorized_keys ' + \
        username + '@' + env.host_string + ':~/ssh')

```

3. Within `fabfile.py` edit line 17 with the root password used while creating your Linode server.

```

# this is the root password you created with the Linode server
env.password = 'New root password here'

```

4. Next edit line 21 with your server IP address that can be found on the [dashboard of your Linode account](#).

```

# this IP address can be found on the Linode Manager Linodes list
env.hosts = ['192.168.1.1']

```

5. Modify line 24 with your full name for the non-root user.
6. Change the value under `env.ssh_key_dir = '~/fsp-deployment-guide/ssh_keys'` the directory where you created your `prod_key` and `prod_key.pub` files.

## Running the Fabric File

Fabric relies on Python having the `fabric` library installed. We'll use a `virtualenv`, a Python tool for isolating packages, to create a separate Python installation that keeps our Fabric package dependency separate from other existing Python packages.

Note that we will have to use Python 2 for now because unfortunately Fabric and Ansible (which we'll start using in the next chapter) [do not yet support Python 3](#).

Create a virtualenv with the following commands. This virtualenv will hold our Python [application dependencies](#). If you already have a directory where you keep your virtualenvs, you can skip the first step and place your new virtualenv in that existing directory.

```
mkdir ~/Envs/
virtualenv -p python2.7 ~/Envs/fspdeploy
source ~/Envs/fspdeploy/bin/activate
```

When you enter the last of those commands your prompt will change to look something like this, which means we've activated the virtualenv in our current shell:

```
(fspdeploy)$
```

If we try to use the `fabric` command right now, it'll fail with the following error message.

```
-bash: fabric: command not found
```

We need to install Fabric into the virtualenv. Use the following pip command to install Fabric:

```
pip install fabric==1.10.2
```

Make sure you've plugged in your server settings at the top of the Fabric file as specified in the previous section. Execute the script from the local command line with `fab bootstrap`. You'll be prompted for a password for the new user and then the script will connect to the server again with that user to complete the steps we walked

through manually earlier this chapter.

Note that this script does not install [fail2ban](#) like we did in the manual steps above. We'll install that package as part of the Ansible scripts we're going to create in the next chapter.

## Next: Operating System

We've secured our server against brute-force password attacks on the root account plus some basic precautions with a public-private key pair. These initial basic security settings are a decent start over the default server settings.

In the next chapter we'll establish the operating system configuration for running Python web applications. We will also introduce Ansible which will provide the rest of the automation necessary for the deployment.

## Additional Server Resources

Numerous additional resources exist if you're not yet entirely comfortable with the steps we performed in this chapter. Be sure to take a look at the following links for learning reinforcement on initial server setup.

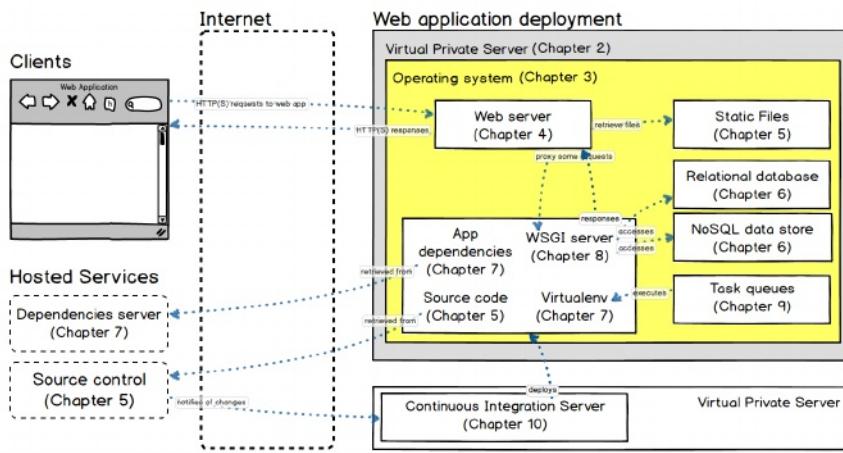
- See the [deployment](#) and [servers](#) pages on Full Stack Python for a slew of additional resources and explanations on these topics.
- [Choosing a low cost VPS](#) reviews the factors that you should weigh when deciding on hosting providers.
- [First 5 Minutes on a Server](#) explains what steps are necessary for a basic security profile on a new server.

**Chapter 3**

# **Operating Systems**

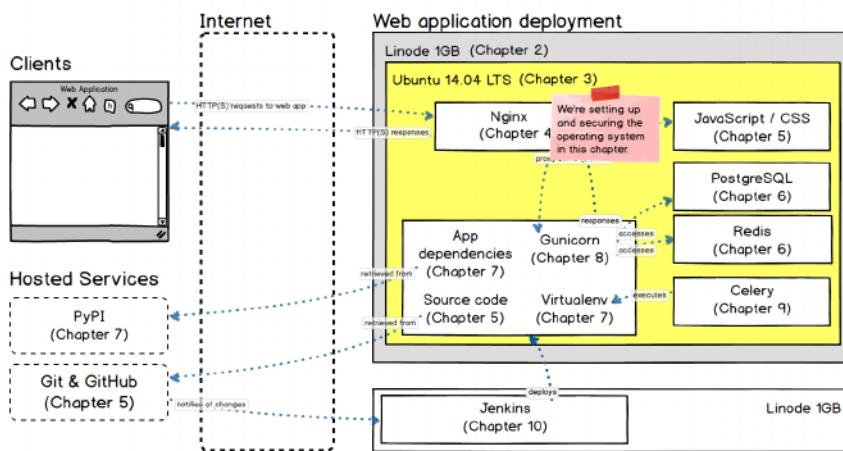
# Operating Systems

An operating system runs on a server and controls access to computing resources that our web application will use to run.



An operating system makes many of the computing tasks we take for granted easy. For example, the operating system enables writing to files, communicating over a network and running multiple programs at once. Otherwise you'd need to control the CPU, memory, network, graphics card and many other components with your own low-level implementation.

In this chapter we'll set up and configure our operating system, Ubuntu Linux 14.04 Long Term Support (LTS), on our Linode virtual private server.



Ubuntu includes a package manager for installing necessary system libraries that we need to run our Python web application. We will also lock down the operating system against unauthorized access attempts.

## Ubuntu

Ubuntu is a Linux distribution packaged by the [Canonical Ltd](#) company. For desktop versions of Ubuntu, GNOME (until the 11.04 release) or Unity (11.10 through current) is bundled with the distribution to provide a user interface.

Ubuntu [Long Term Support](#)(LTS) releases are the recommended versions to use for deployments. LTS versions receive five years of post-release updates from Canonical. Every two years, Canonical creates a new LTS release, which allows for an easy

upgrade path as well as flexibility in skipping every other LTS release if necessary. As of 2015, [14.04 Trusty Tahr](#) is the latest LTS release.

Mac OS X, and to a lesser extent Windows, are fine for developing your Python application. However, Windows and Mac OS X are not appropriate for our production deployment.

## Installing Ubuntu Packages

Ubuntu uses the Debian distribution as a base for packages, including the [apt package manager](#). There are several required [apt](#) packages found on Linux servers running a Python stack. These packages are:

- [python-dev](#) for header files and static library for Python
- [python-virtualenv](#) for creating and managing Python [virtualenvs](#) to isolate library dependencies
- [git-core](#) for the [Git](#) version control system.
- [nginx](#) is a web server that will answer our incoming HTTP requests.
- [supervisor](#) will control the state of our WSGI server and other installed programs

Let's install these packages that we'll need throughout our deployment.

1. First ensure the package manager's local cache is up to date. We performed this in the previous chapter to properly install [fail2ban](#) but it's always a good practice to update just before installing new packages.

```
$ sudo apt-get update
```

2. Next install the required packages for our Python web application deployment.  

```
$ sudo apt-get install fail2ban python-virtualenv python-dev
```
3. It'll take a little while for the packages to get installed since Aptitude is downloading everything from the central Ubuntu package repositories.
4. When it's done you'll see something like the following line and you'll be back at the input prompt.

```
Processing triggers for libc-bin (2.19-0ubuntu6) ...
```

With these packages in place we now have the basic system dependencies to get our Python environment running. However, our system still needs a firewall to lock down ports other than 22 (ssh), 80 (HTTP) and 443 (HTTPS).

## Enable the Firewall

Ubuntu has a handy tool named Uncomplicated Firewall ([ufw](#)). We'll run a few commands with [ufw](#) to stop Internet traffic against ports we don't intend to expose to the outside world.

```
sudo ufw allow 22  
sudo ufw allow 80  
sudo ufw allow 443
```

The first three above commands tell our operating system to only allow network connections to go through to ports 22, 80 and 443. We now need to enable the firewall with the following command:

```
sudo ufw enable
```

Our firewall is in place, but it was a pain to manually run all the above commands. Let's now take a look at the tool Ansible which will help us automate our deployment process.

## Ansible

[Ansible](#) is an open source configuration management tool written in Python that will allow us to automate the steps we performed manually above. We'll also build on the Ansible scripts, which are called ["playbooks"](#) in Ansible terminology, in each subsequent chapter.

Ansible has hundreds of built-in modules to execute tasks for setting up and running servers. The source code for every module is freely available on Ansible's [GitHub repository](#).

## Automating Operating System Setup

Let's write the structure for the Ansible playbook which we'll fill in throughout each chapter's automation section from now on. If you want to skip typing these instructions in there's an [open source GitHub repository](#) that's [tagged with steps for each chapter](#).

1. In the last chapter we created a `prod` directory for our `fabfile.py` script. Now we're going to add our Ansible files within the same directory. This can either be under your project or a separate source control repository. Remember to keep your confidential information such as usernames and passwords private!

```
cd prod
```

```
touch deploy.yml hosts  
mkdir roles group_vars
```

2. We just created a new file for the main YAML playbook named deploy.yml, which we'll use to coordinate our deployment's various roles. We also created a hosts file that'll tell Ansible what server IP addresses we should deploy to.
3. Fill in the `prod/deploy.yml` file with the following text.

```
###  
# This playbook deploys the whole application stack in this site.  
##  
- name: apply common configuration to server  
  hosts: all  
  user: deployer  
  roles:  
    - common
```

4. Fill in the hosts file with the following text. Replace `192.168.1.1` with your Linode server's IP address.

```
[common]  
192.168.1.1
```

5. Next we're going to fill in the roles that will automate the steps we otherwise would have to manually execute. Some of these directories will be used later to set up the database and web server.

```
mkdir -p roles/common/tasks  
cd roles  
mkdir -p common/handlers common/templates
```

6. We need to set variables to complete the roles. The variables won't be used until we fill out roles that have the variable tokens in them.

```
touch group_vars/all
```

7. Create the following YAML within the `group_vars/all` file.

```
# Chapter 3: Operating System (Ubuntu)
app_name: cyoa
deploy_user: deployer
deploy_group: deployers
## this is the local directory with the SSH keys and known_hosts
## file do not include a trailing slash
ssh_dir: ~/fsp-deployment-guide/ssh_keys
```

8. Let's automate the first steps we took on the server where we manually updated the apt repository and installed required system packages. Create a file named `prod/roles/common/tasks/main.yml` and have it contain the following YAML.

```
###  
# configures the server and installs the web application  
##  
- include: ubuntu.yml
```

9. We need to create the `ubuntu.yml` file we just wrote on the last line of `main.yml`. Create a file named `roles/common/tasks/ubuntu.yml` so that it is in the same directory as the `main.yml` file and fill it with the following YAML lines.

```
##  
# updates the APT package cache and install packages  
# servers necessary for web. also enables firewall  
##  
- apt: update_cache=yes  
  sudo: yes  
  
- name: ensure web server packages are installed  
  apt: name={{item}}  
  sudo: yes  
  with_items:  
    - fail2ban  
    - python-virtualenv  
    - python-dev  
  
- name: enable SSH in firewall
```

```
ufw: rule=allow port=22
- name: enable HTTP connections for web server
  ufw: rule=allow port=80
- name: enable HTTPS connections for web server
  ufw: rule=allow port=443
- name: enable firewall
  ufw: state=enabled
```

10. We need to install Ansible into our virtualenv. If the virtualenv isn't activated right now, make sure to activate it first with the following command. Make sure to specify the location where you created the virutalenv back in chapter 2.

```
source ~/Envs/fspdeploy/bin/activate
```

11. Use the following pip command to install Ansible:

```
pip install ansible==1.9.2
```

12. We've created the basic outline of an Ansible playbook that automates what our manual instructions set up so far. In order to run the playbook I recommend creating a shell script named `deploy_prod.sh` in the base directory of our project with the following contents.

```
#!/bin/bash
ansible-playbook ./prod/deploy.yml --private-key=\
./ssh_keys/prod_key -K -u deployer -i ./prod/hosts
```

13. Be sure to setthe appropriate permissions on the file.

```
chmod 0744 deploy_prod.sh
```

14. Now we can invoke that file by running it on the command line as follows.

```
./deploy_prod.sh
```

15. You'll be asked for the sudo password once to kick things off then if everything's been typed in properly this playbook will automate the manual steps from this chapter and the previous one.

## Next: Web Server

We'll continue building our Ansible playbook out in the following chapters. In the next chapter on web servers we can configure the Nginx web server which was just installed through a system package.

## Additional Operating System Resources

The following references provide greater insight into Linux distributions and how to deploy on Ubuntu.

- [What is a Linux distribution and how do I choose the right one?](#) will explain more about what distributions are.
- Lifehacker's [guide to choosing a Linux distro](#) explains more about Ubuntu versus other potential Linux distributions.
- [Choosing a Linux Distribution](#) is Rackspace's detailed guide to Linux distributions.
- Digital Ocean has a detailed [walkthrough on setting up Python web applications](#) specifically for Ubuntu.

## Additional Ansible Resources

Ansible is a powerful configuration management tool. Once you get over the initial YAML-learning hump, it's easy to deploy an entire application stack, as we'll show throughout the rest of this book. The following resources will help you get more comfortable with using Ansible.

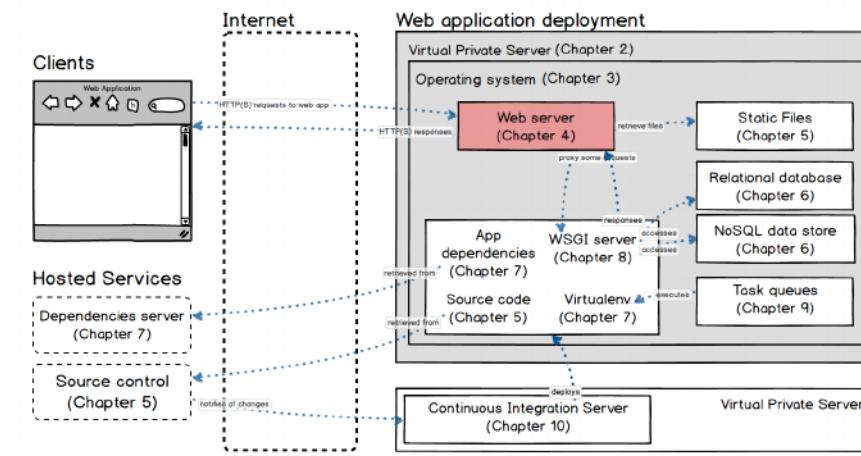
- The [official Ansible documentation](#) is a good first link to check out.
- The [Ansible Briefs](#) email newsletter by Matt Jaynes always has the latest helpful resources and tutorials on Ansible.
- [Ansible vs. Shell Scripts](#) explains why automating with Ansible instead of shell scripts is a good idea.
- This [\(Detailed\) Introduction to Ansible](#) is a very helpful first steps with Ansible guide.
- [Ansible Text Message Notifications with Twilio SMS](#) is my blog post with a detailed example for using the [Twilio module](#) that's included as part of core Ansible 1.6+.
- [First Five \(and a half\) Minutes on a Server with Ansible](#) is similar to the first five minutes on a server post but shows how to automate those steps using Ansible.

**Chapter 4**

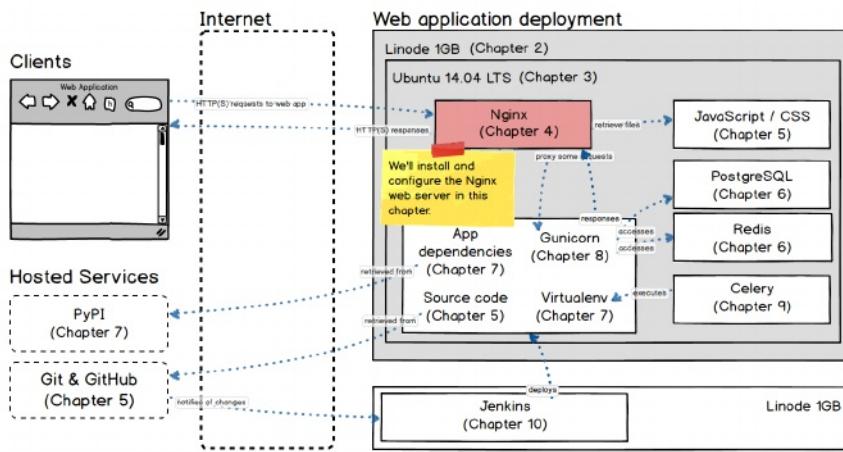
# **Web Servers**

# Web Servers

Web servers respond to [Hypertext Transfer Protocol](#) (HTTP) requests from clients and send back a response containing a status code and content such as HTML, XML or JSON as well. In our deployment, the web server handles serving up static files like our CSS and JS, while serving as a proxy that passes requests to the WSGI server and the response back to the web browser.



We'll use the web server [Nginx](#) (pronounced "Engine-X") throughout our deployment.



Nginx is used by many of the top websites on the Internet. Its configuration files are generally considered easier to read and write than the Apache HTTP server's config files.

## Web Servers in Our Deployment

The web server in our deployment will perform two roles:

1. Handle requests for static files and respond with appropriate files
2. Reverse proxy requests and responses to and from our WSGI server

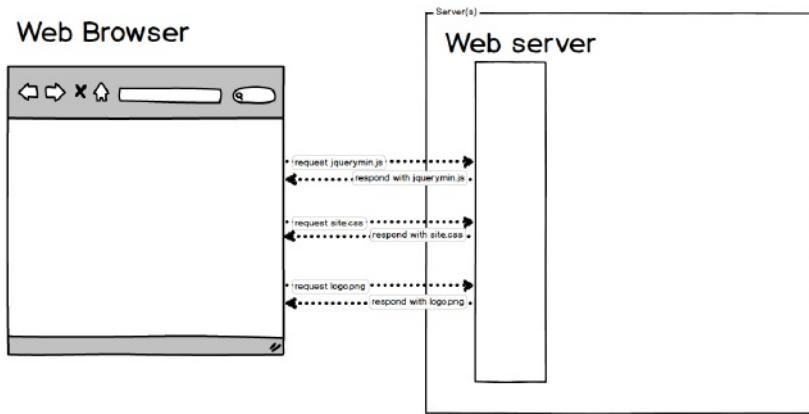
In the first use case our Nginx web server will directly handle requests for static files by responding with appropriate files from the `static/` directory. In the `static/` directory

there will be subfolders for various types of static files such as JavaScript, CSS and images.

For the second use case the web server be a reverse proxy to pass on requests to our WSGI server so it can run our Python code and craft the response. A reverse proxy is simply a middleman that accepts an incoming HTTP request and passes it along to the WSGI server so it can process the request and issue a response.

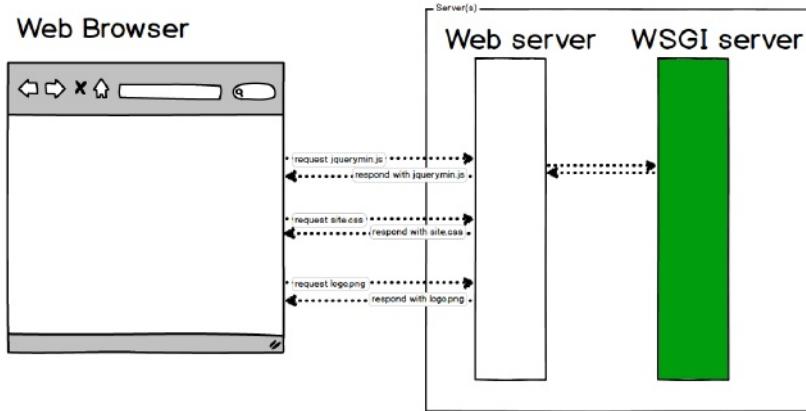
## Visualizing our Web Server's Purpose

Let's visualize how our web server is working in the first and second use cases.



The web server sends files to a web browser based on the browser's requests. In the first request, the browser asked for the file `jquery.min.js` and the server responded with the JavaScript file. The web browser then accessed `site.css` and `logo.png` and again the server responded with those static files.

However, not all requests are for static files. For example, a page from our web application must be generated by the WSGI server running our Python code. That scenario is shown in the following image.



In this case the web server is configured to pass requests for non-static files to the WSGI server. We'll lay the groundwork for that configuration in this chapter and finish it off later in the WSGI server chapter.

## Nginx

In our deployment we'll use the Nginx web server, the second most commonly used web server after Apache. While both Nginx and Apache are fantastic pieces of software, we're going to use Nginx because it's easier to install and configure if you've never done this type of thing before.

Nginx is used at the largest scale by many organizations on the web so you won't have to worry about swapping it out later for Apache.

Before we install and configure Nginx on our server we'll need to perform two steps: set a domain name to point to our server and (optionally) create an SSL certificate.

We'll first take care of setting up the domain name then give you the choice to create an SSL certificate to provide an HTTPS connection or just use plain old HTTP to serve the application.

## Installing Nginx

We need to use the system package manager to install Nginx just as we did with `fail2ban` and our Python packages.

1. First install the `nginx` package via apt on our remote machine.

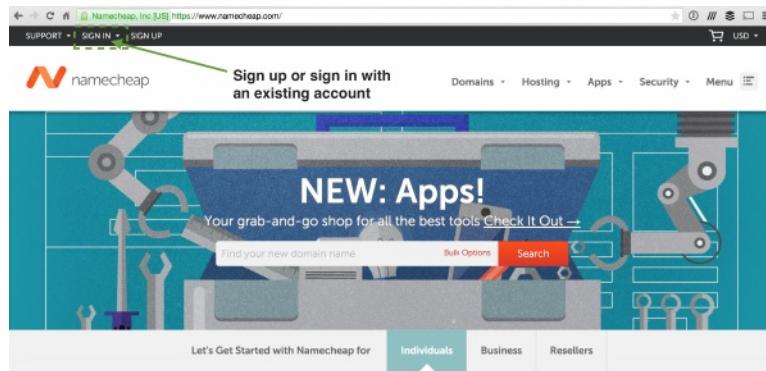
```
sudo apt-get install nginx
```

Next we need to set up our domain name so we can access the app from a URL instead of the IP address. We'll use Namecheap to handle the domain name server resolution.

## Domain Name Service Resolution

For our application we don't want to send users a raw IP address. Instead, we want a domain name so we can access our application through a normal looking URL. In this case, we'll see how to use a domain name registered with Namecheap to point our application to `cyoa.mattmakai.com`

1. Start by going to <https://www.namecheap.com> in your browser. Sign up for an account or sign in to your existing account.



Create your pro web presence in no time.

2. Once you've signed in, go to manage domains.

The screenshot shows the 'Member's Home' page for Matthew. It features a sidebar with account statistics: 3 domains expiring within 30 days, 0 domains listed for sale, 0 pending offers, 0 WhoisGuard subscriptions, 0 unused WhoisGuard, 17 unread messages, 0 transfer requests, 0 private email domains, 0 unread privacy policies, and 0 Onepager subscriptions. A green arrow points from the 'Click to view your purchased domains' link to the 'Purchased Domains' section in the sidebar. The main content area has sections for 'Apps' (with a 'FUND YOUR ACCOUNT' button) and 'More Links' (including 'Your Domains / Products', 'Order History', 'Transfer History', 'Get EFF Code', 'Namecheap Deals', 'Contact Us', and 'Help').

- Select the domain name you want to set up for this application.

The screenshot shows the Namecheap 'Your Domains' page. On the left, there's a sidebar with various service links like 'Domain Names', 'Hosting, Email & Apps', and 'Transfers'. The main area is titled 'Your Domains' and contains a table of domains. One row for 'mattheekay.com' is highlighted with a green dashed border. A green arrow points from the text 'Click the link to the domain in your list that you want to use' to this highlighted row. The table has columns for 'DOMAIN NAME', 'CREATED ON', 'EXPIRES', and 'STATUS'.

DOMAIN NAME	CREATED ON	EXPIRES	STATUS
<a href="#">mattheekay.com</a>	Sun 03/23/2014	Wed 03/23/2016	<input checked="" type="checkbox"/>
Tue 04/15/2014	Fri 04/15/2016	<input checked="" type="checkbox"/>	
Tue 04/15/2014	Fri 04/15/2016	<input checked="" type="checkbox"/>	
Tue 04/29/2014	Fri 04/29/2016	<input checked="" type="checkbox"/>	
Sun 12/13/2012	Fri 12/23/2016	<input checked="" type="checkbox"/>	
Sun 03/23/2014	Wed 03/23/2016	<input checked="" type="checkbox"/>	
Fri 06/13/2014	Mon 06/13/2016	<input checked="" type="checkbox"/>	
Tue 06/04/2013	Thu 06/04/2015	<input checked="" type="checkbox"/>	
Fri 02/07/2014	Sun 02/07/2016	<input checked="" type="checkbox"/>	
Mon 01/02/2012	Sat 11/26/2016	<input checked="" type="checkbox"/>	

- Click "All Host Records".

**General**

-  Change Contacts
-  Transfer DNS to Webhost
-  Switch To DNS System v1
-  Registrar Lock
-  Auto-Renew
-  List Domain For Sale
-  TypoScan

**Websites**

-  Onepager Website

**E-mail Management**

-  E-mail Forwarding Setup
-  Private Email Hosting

**Host Management**

-  URL Forwarding
-  URL Frame Meta Tags
-  All Host Records

**Click this link  
for host  
records on  
the side bar**



5. There are two choices here. If you want the standard `www` in your URL, set `www` with your server's IP address and specify it as an A (Address) record. If instead you want a different subdomain, such as `yoa` as I've done here, then enter the subdomain with the server's IP address and select the A (Address) record in the dropdown.



6. Save the newly entered information. It may take a few minutes to take effect as the data must be propagated to many Domain Name System hosts.

Now we can configure Nginx to either use HTTP or HTTPS. Both options are included in this chapter. If you just want to deploy without encryption provided by HTTPS, read the next section, but know that your application won't be secure. If you want security and encryption provided by the latest version of SSL/TLS then skip down to "Configure Nginx with HTTPS".

## Configure Nginx Without HTTPS

In this section we'll configure Nginx without HTTPS provided by SSL/TLS. If you want your application to use encryption (and it should!), skip this section and move onto the next one named "Configure Nginx with HTTPS".

1. Create a configuration file under `/etc/nginx/conf.d` with your app name and .conf extension. In this case we'll deploy with the `cyoa.conf` filename, so the full path and filename will be `/etc/nginx/conf.d/cyoa.conf`. Note that we'll need to use sudo privileges to create and save the file within Nginx's configuration directory. Add the following lines to the file.

```

upstream app_server_wsgiapp {
    server localhost:8000 fail_timeout=0;
}

server {
    listen 80;
    # make sure to change the next line to your own domain name!
    server_name cyoa.mattmakai.com;
    access_log      /var/log/nginx/cyoa.access.log;
    error_log       /var/log/nginx/cyoa.error.log info;
    keepalive_timeout 5;

    # nginx serve up static files and never send to the WSGI server
    location /static {
        autoindex on;
        alias /home/deployer/cyoa/static;
    }

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        if (!-f $request_filename) {
            proxy_pass http://app_server_wsgiapp;
            break;
        }
    }

    # this section allows Nginx to reverse proxy for websockets
    location /socket.io {
        proxy_pass http://app_server_wsgiapp/socket.io;
        proxy_redirect off;
        proxy_buffering 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_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }
}

```

}

That's all the configuration you need to use Nginx to serve static files and provide a reverse proxy to the WSGI application. Skip down to the "Restarting with a new configuration" section to make the configuration take effect.

## Create an SSL Certificate

We're first going to create the SSL certificate before we set up Nginx on our remote server. The certificates will get uploaded manually and then we'll modify the Ansible playbook to handle the step in the future.

Next we create a self-signed certificate. That means this certificate will give a warning on most browsers if you access the web application because the certificate has not been blessed by a central certificate authority. I recommend buying a certificate from a Certificate Authority (CA) such as [Digicert](#) or [StartSSL](#) once you're ready to go live. Replace this self-signed certificate with the certificate from a valid CA and those browser warnings will go away.

If you have trouble with the following steps be sure to read this [detailed guide on creating a self-signed certificate](#)

**Execute these commands on your local machine.** You'll need OpenSSL installed which is part of Mac OS X and most Linux distributions.

1. Create a subdirectory from our base directory to hold our SSL certificate files as we generate them.

```
mkdir ssl_cert  
cd ssl_cert
```

2. Generate a private key file using the `openssl` command as follows.

```
# write a private key file using a 4096 bit key
openssl genrsa -des3 -out ssl.key 4096
```

3. `ssh-keygen` will tell you that an RSA private key is being generated. Then you'll be prompted for a passphrase as shown below. Enter a passphrase.

```
# enter a pass phrase between 8-16 characters
Enter pass phrase for ssl.key:
# verify the pass phrase by entering it again
Verifying - Enter pass phrase for ssl.key:
```

4. Now you have an `ssl.keyfile` in the current directory. Next we need to generate a certificate signing request (CSR).

```
# openssl allows us to create a signing request file
openssl req -new -key ssl.key -out ssl.csr
```

5. Fill in the information to the questions with your information. My example information is shown below. Not all questions need to be answered. Some of the questions, such as "Organizational Unit Name" can simply be left blank.

```
Enter pass phrase for ssl.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Full Stack Python
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.fullstackpython.com
Email Address []:matthew.makai@gmail.com

Please enter the following 'extra' attributes
to be sent with your certificate request
```

```
A challenge password []:
An optional company name []:
```

6. Remove the passphrase from the key file because it's a pain when using automation to start and stop the web server as the passphrase would need to be entered for all operations.

```
# recreate the key file but without the pass phrase
openssl rsa -in ssl.key -out cyoa.key
```

7. There is no passphrase on `cyoa.key` and now we can finally generate the self-signed certificate file.

```
# this ssl.crt is the file we actually care about. it'll have a
# lifetime of 730 days (about 2 years)
openssl x509 -req -days 730 -in ssl.csr -signkey cyoa.key -out cyoa.crt
```

We'll upload the `cyoa.key` and `cyoa.crt` files to the server as we configure our Nginx server in the next section.

## Install SSL Certificates

Nginx was already installed on our server in chapter two when we executed `apt-get install nginx`. Now we need to configure it to serve up static files as well as pass requests to the WSGI server that we will set up later.

1. **Log back into your remote server.** Create a directory to store the SSL certs temporarily on the remote server.

```
mkdir /home/deployer/ssl_tmp/
```

2. **Run the following command from your local machine from within the `ssl_cert` directory you previously created.** From our local machine we need to copy in the SSL keys we just generated. Within the directory that stores

the SSL key files, run the following commands to get them securely copied to the server. Remember to use the IP address of your server instead of the one shown in this command.

```
scp -i ..ssh_keys/prod_key cyoa.key cyoa.crt \
deployer@{your.server.ip.address}:/home/deployer/ssl_tmp/
```

3. **Perform the following commands on the deployment server**Nginx's configuration can only be changed with root privileges so with the deploy user first execute this command to ensure the rest of these steps are performed with superuser privileges.

```
# -i argument allows us to interactively use multiple sudo commands
sudo -i
```

4. Change into the Nginx configuration directory.

```
cd /etc/nginx/
```

5. Create an SSL directory for our Nginx web app.

```
mkdir /etc/nginx/cyoa
```

6. Copy the SSL certificate files from the home directory.

```
cp /home/deployer/ssl_tmp/* /etc/nginx/cyoa/
```

7. Delete the SSL certificates from the home directory.

```
rm -rf /home/deployer/ssl_tmp/
```

We now need to set up Nginx's configuration so that it uses the SSL certificates. We'll also prepare it for passing requests to the WSGI server that will be set up in a later chapter.

# Configure Nginx with HTTPS

Now we'll configure Nginx using a custom configuration file. This configuration file will be set up for our SSL connection and our WSGI server we'll set up later in the book.

1. Create a configuration file under `/etc/nginx/conf.d` with your app name and .conf extension. In this case we'll deploy with the `cyoa.conf` filename, so the

full path and filename will be `/etc/nginx/conf.d/cyoa.conf`. Note that we'll need to use sudo privileges to create and save the file within Nginx's configuration directory. Add the following lines to the file.

```
upstream app_server_wsgiapp {
    server localhost:8000 fail_timeout=0;
}

server {
    listen 80;
    # make sure to change the next line to your own domain name!
    server_name cyoa.mattmakai.com;
    rewrite ^(.*) https://$server_name$1 permanent;
}

server {
    # make sure to change the next line to your own domain name!
    server_name cyoa.mattmakai.com;
    listen 443 ssl;
    ssl_certificate /etc/nginx/cyoa/cyoa.crt;
    ssl_certificate_key /etc/nginx/cyoa/cyoa.key;
    ssl_session_timeout 1d;
    ssl_session_cache shared:SSL:50m;
    ssl_protocols TLSv1.1 TLSv1.2;
    ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-
SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-
GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGM::ECDHE-RSA-AES128-SHA256:ECDHE-
ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-
SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-
RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-
SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-
SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-
SHA:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-
RSA-DES-CBC3-SHA:!KRB5-DES-CBC3-SHA';

ssl_prefer_server_ciphers on;
```

```

access_log /var/log/nginx/cyoa.access.log;
error_log /var/log/nginx/cyoa.error.log info;
keepalive_timeout 5;

# nginx serve up static files and never send to the WSGI server
location /static {
    autoindex on;
    alias /home/Deployer/cyoa/static;
}

location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    if (!-f $request_filename) {
        proxy_pass http://app_server_wsgiapp;
        break;
    }
}

# this section allows Nginx to reverse proxy for websockets
location /socket.io {
    proxy_pass http://app_server_wsgiapp/socket.io;
    proxy_redirect off;
    proxy_buffering 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_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
}
}

```

Now Nginx is configured with the SSL certificate we created in the earlier section and will only use secure methods to communicate with clients. Before the application

works though, we'll have to restart with a new configuration. We handle the restarting of Nginx in the next section.

## Restarting Nginx

We need to restart the Nginx server to enable our new configuration, regardless of whether it's using SSL/TLS or not.

1. Delete the default website file if it exists.

```
sudo rm /etc/nginx/sites-enabled/default
```

2. Restart Nginx so the new configuration takes effect.

```
sudo service nginx restart
```

Our web server Nginx is now manually configured to serve up static files such as JavaScript, CSS and images. Nginx will also pass requests up to the WSGI server that's running our Python application, including websocket connections. However, before we get the WSGI server running we will need to grab our web application's source code. That's the subject of the next chapter.

If you're running through the manual steps first you can now flip to the next chapter on source control. To learn how to automate the Nginx configuration steps we just performed, keep reading below.

## Automating the Nginx Configuration

We need to add new files to our Ansible playbook to automate the Nginx configuration. Since this chapter had the option of using either plain old HTTP or

HTTPS, we will create a flag in Ansible that can be flipped to either set up the server with HTTP or HTTPS.

1. Open `prod/group_vars/all` and append the following lines to the existing file.  
These are new variables we'll need for the automated deployment.

```
# Chapter 4: Web Servers
## deploy_ssl true for HTTPS, false for only HTTP
deploy_ssl: true
wsgi_server_port: 8000
fqdn: cyoa.mattmakai.com
app_dir: /home/{{deploy_user}}/{{app_name}}
## local directory SSL certificates should be located here
ssl_certs_dir: ~/fsp-deployment-guide/ssl_cert/
```

2. Create a new file named `prod/roles/common/tasks/nginx_ssl.yml`. Ensure the file has the follow contents, which install and configure Nginx for HTTPS connections.

```
##
# Configuration for Nginx web server
##
- name: ensure Nginx is installed via the system package
  apt: name=nginx state=present update_cache=yes
  sudo: yes

- name: create nginx ssl directory if it does not exist
  file: path=/etc/nginx/{{ app_name }} state=directory
  sudo: yes

- name: write SSL certificate file
  copy: src={{ ssl_certs_dir }}/{{ app_name }}.crt
    dest=/etc/nginx/{{app_name}}/{{app_name}}.crt
  sudo: yes

- name: write SSL key file
  copy: src={{ ssl_certs_dir }}/{{ app_name }}.key
    dest=/etc/nginx/{{ app_name }}/{{ app_name }}.key
  sudo: yes
```

```

- name: ensure default symbolic linked website is deleted
  file: path=/etc/nginx/sites-enabled/default state=absent
  sudo: yes

- name: write SSL nginx configuration
  template: src=nginx_ssl.conf.j2
    dest=/etc/nginx/conf.d/{{ app_name }}.conf
  sudo: yes
  notify:
    - restart nginx

```

3. Create another new file named `prod/roles/common/tasks/nginx_no_ssl.yml` with the following YAML. This file is used to install and configure Nginx when HTTPS is not necessary.

```

###  

# Configuration for Nginx web server without HTTPS  

##  

- name: ensure Nginx is installed via the system package
  apt: name=nginx state=present update_cache=yes
  sudo: yes

- name: ensure default symbolic linked website is deleted
  file: path=/etc/nginx/sites-enabled/default state=absent
  sudo: yes

- name: write non-SSL nginx configuration
  template: src=nginx_no_ssl.conf.j2
    dest=/etc/nginx/conf.d/{{ app_name }}.conf
  sudo: yes
  notify:
    - restart nginx

```

4. Create a file named `prod/roles/common/templates/nginx_ssl.conf.j2` that contains the following lines.

```

upstream app_server_wsgiapp {
  server localhost:{{ wsgi_server_port }} fail_timeout=0;
}

```

```

server {
    listen 80;
    server_name {{ fqdn }};
    rewrite ^(.*) https://$server_name$1 permanent;
}

server {
    server_name      {{ fqdn }};
    listen          443 ssl;
    ssl_certificate /etc/nginx/{{ app_name }}/{{ app_name }}.crt;
    ssl_certificate_key /etc/nginx/{{ app_name }}/{{ app_name }}.key;
    ssl_session_timeout 1d;
    ssl_session_cache shared:SSL:50m;
    ssl_protocols     TLSv1.1 TLSv1.2;
    ssl_ciphers       'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-
SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-
GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGM::ECDHE-RSA-AES128-SHA256:ECDHE-
ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDSA-ECDSA-AES128-SHA:ECDHE-RSA-AES256-
SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDSA-ECDSA-AES256-SHA:DHE-
RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-
SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-
SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-
SHA:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-
RSA-DES-CBC3-SHA:!KRB5-DES-CBC3-SHA';
    ssl_prefer_server_ciphers on;
    access_log        /var/log/nginx/{{ fqdn }}.access.log;
    error_log         /var/log/nginx/{{ fqdn }}.error.log info;
    keepalive_timeout 5;

# nginx serve up static files and never send to the WSGI server
location /static {
    autoindex on;
    alias {{ app_dir }}/{{ app_name }}/static;
}

location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    if (!-f $request_filename) {
        proxy_pass http://app_server_wsgiapp;
        break;
    }
}

```

```

# this section allows Nginx to reverse proxy for websockets
location /socket.io {
    proxy_pass http://app_server_wsgiapp/socket.io;
    proxy_redirect off;
    proxy_buffering 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_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
}
}

```

5. Create a file named `prod/roles/common/templates/nginx_no_ssl.conf.j2` that contains the following lines to set up the web server when no HTTPS connections are desired.

```

upstream app_server_wsgiapp {
    server localhost:{{ wsgi_server_port }} fail_timeout=0;
}

server {
    listen 80;
    server_name {{ fqdn }};
    access_log  /var/log/nginx/{{ fqdn }}.access.log;
    error_log   /var/log/nginx/{{ fqdn }}.error.log info;
    keepalive_timeout      5;

    # nginx serve up static files and never send to the WSGI server
    location /static {
        autoindex on;
        alias {{ app_dir }}/{{ app_name }}/static;
    }

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
    }
}

```

```

proxy_redirect off;
if (!-f $request_filename) {
    proxy_pass http://app_server_wsgiapp;
    break;
}

# this section allows Nginx to reverse proxy for websockets
location /socket.io {
    proxy_pass http://app_server_wsgiapp/socket.io;
    proxy_redirect off;
    proxy_buffering 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_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
}
}

```

6. The above configuration is a template with tokens that will be replaced with values in variables from the group\_vars directory when the Ansible playbook is executed. The result is a file on our server that contains the same configuration file we otherwise would have to write manually.
7. Open and update `prod/roles/common/tasks/main.yml`. We're adding logic to this file so based on the `deploy_ssl` variable Ansible will configure Nginx either with or without HTTPS connections. Append the following YAML to the existing file.

```

- include: nginx_ssl.yml
  when: deploy_ssl

- include: nginx_no_ssl.yml
  when: not deploy_ssl

```

8. There is one more file to create before Ansible will have what it needs to automate the set up of the web server. Create a file named `main.yml` within the `prod/roles/common/handlers` directory. Write the following contents into the file. This handler allows us to restart Nginx during the playbook execution when it's necessary such as upon a configuration file modification.

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

9. Now we can run our Ansible playbook again with our shell script to install and configure Nginx on our server.

```
./deploy_prod.sh
```

Nginx is ready to serve HTTP requests but we need to run the WSGI server and our app to handle anything other than static files.

## Next: Source Control

In the next chapter we'll use Git to clone our source repository and ensure the appropriate code is on our server for the rest of the deployment.

## Additional Web Server Resources

There are numerous guides that explain the underlying concepts web servers implement as well as how Nginx works.

- The [HTTP/1.1 Specification](#) is the official spec for how the current version of HTTP works. Version 2.0 is slowly replacing 1.1 but for now this is the spec to read.

- Ars Technica has a detailed guide for [How to set up a safe and secure Web server](#).
- [Nginx web server tutorials](#) are groups of tutorials for setting up and configuring Nginx.
- [Nginx for Developers: An Introduction](#) provides exactly what developers need to start using Nginx.
- An example of an [Nginx security configuration](#) sheds further light on settings you may want to tweak.
- A reference with the full list of [HTTP status codes](#) is provided by W3C.
- An optimization guide for [battle ready Nginx](#) is a well done tutorial.
- [A faster Web server: ripping out Apache for Nginx](#) explains how Nginx has replaced Apache in some environments.
- Mozilla has a tool for [creating SSL/TLS certificate configurations for web servers](#) in case you want to check your configuration against a recommended one.

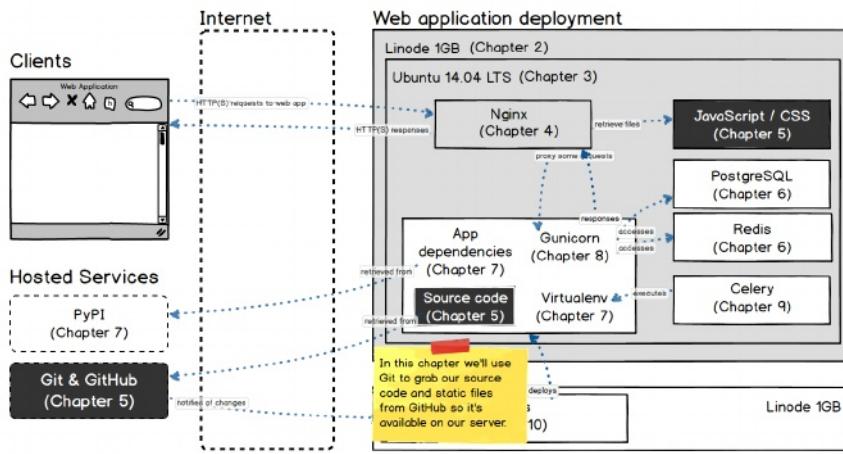
Chapter 5

# Source Control

# Source Control

Source control, also known as *version control*, stores source code files along with metadata on changes, such as character additions and deletions.

In an application deployment, source control can provide a bridge between the current production-ready version of the code stored on the server and the updates that need to be applied as the application continues to be worked on.

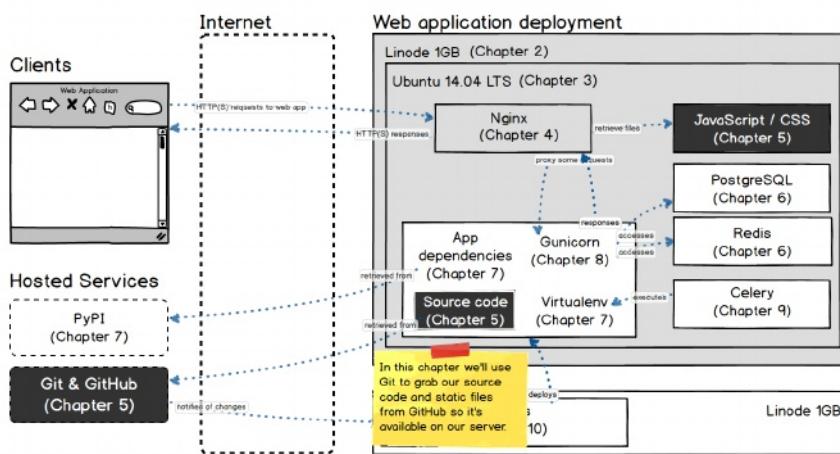


Version control systems allow developers to modify code without worrying about permanently screwing something up. Unwanted changes can be rolled back to a previous working code state.

Source control also eases developing software for teams. One developer can combine her code modifications with other developers' code by viewing differences in the code and merging in appropriate changes.

Note that some developers recommend deployment pipelines package the source code to deploy it and never have a production environment touch a source control system directly. However, for small scale deployments it's often easiest to pull from source code when you're getting started instead of figuring out how to wrap the Python code in a system installation package.

Our deployment will use the source control deployment approach for getting our code onto the production server. This method for deploying code via the `git clone` and `git pull` commands will make our deployment process look like the following diagram.



For our deployment we'll be using Git as our source control system and GitHub as a remote hosted source control service. In this chapter, we'll install Git, set up a read-only deploy key and use Git to transfer our source code and static files onto the server.

# Hosted Source Control Services

We could install Git on a remote server and use that server to back up our source control, but it's easier to get started with a hosted version control service. You can transition away from the service at a later time by moving your repositories to your own server if your needs change. A couple of recommended hosted version control services are:

- [GitHub](#) is currently the most commonly used Git source control platform.
- [BitBucket](#) provides free Git repositories for public projects and private repositories for up to five users.

As mentioned earlier, we'll use GitHub during this deployment. Sign up for a free GitHub account now as we'll use it throughout this chapter.

## Creating a Deploy Key

We need to create a deploy key separate from the key we use to log into the server. This deploy key will grant read-only access from our server to the single repository that we want to use the commands `git clone` and `git pull` to grab our code.

1. On the remote server create a new directory called `deploy_key`.

```
mkdir ~/deploy_key  
cd ~/deploy_key
```

2. Execute the following command to create a new public-private key pair.

**Perform these commands on your production server.**

```
# this is the same command we performed in the second  
# chapter, but we're using it to create a deploy key now  
  
# -t specifies the algorithm, -b argument specifies the
```

```
# number of bits to use  
ssh-keygen -t rsa -b 2048
```

3. When prompted for a file in which to save the keys do not use the default.

Instead, enter the following directory and file into the prompt.

```
# save the private key in our current directory  
.deploy_key
```

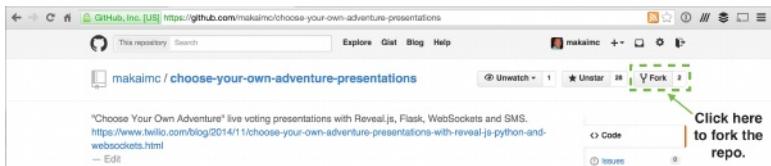
4. Press enter twice when prompted for a passphrase. We won't use a passphrase on the deploy key.
5. Now you again have two new files: a private key `deploy_key` and a public key `deploy_key.pub`. We'll next put the public key into GitHub so we can clone our code repository.

These new deploy keys are different than the `prod_key` and `prod_key.pub` that we created in chapter 2. They cannot be used interchangeably. For example, the `prod_key` is only used to log into our server, whereas the `deploy_key` can clone a Git repository from GitHub but cannot be used to log into our production server.

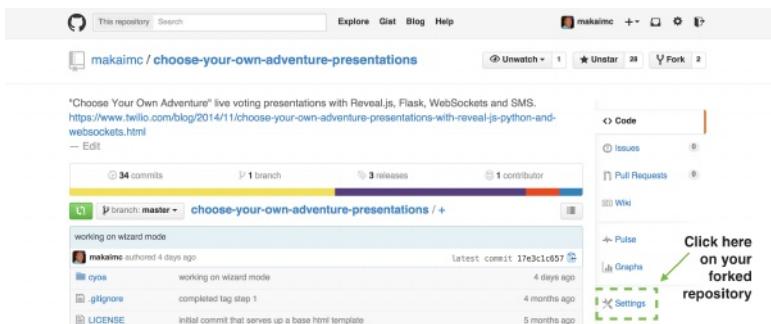
## Authorizing Git Clone Access

To execute the Git repository clone you'll need to fork the [Choose Your Own Adventure Presentations](#) project and add a deploy key. The following steps will walk through how to make that happen.

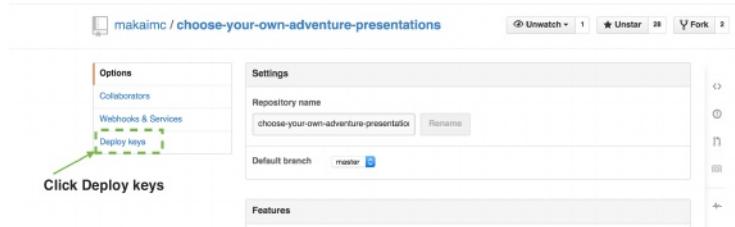
1. Go to the [Choose Your Own Adventure Presentations](#) repository page on GitHub.
2. Press the fork button and select where to fork the repository.



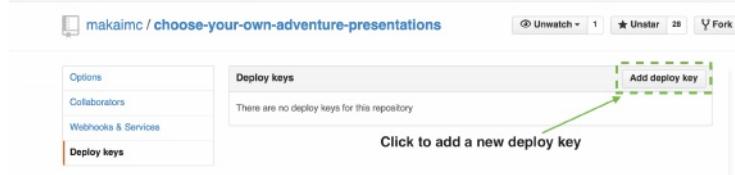
3. When the repository finishes forking the forked version's web page will appear. We need to add the deploy key we just created so that our new server can pull down the code.
4. Click the settings link on the right navigation bar within the forked repository's page.



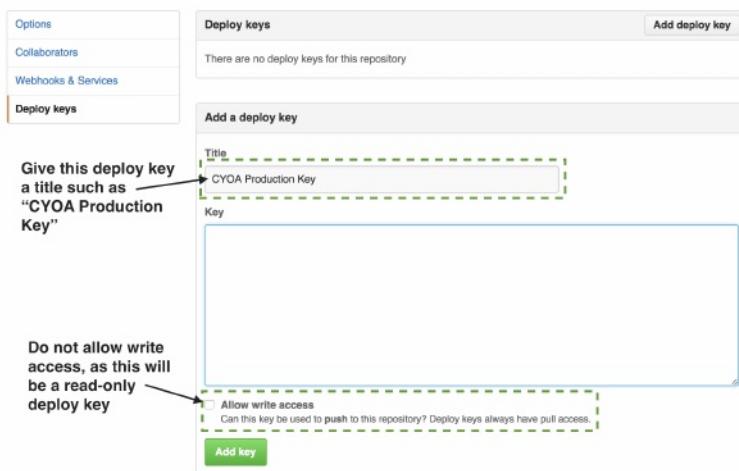
5. Click "Deploy Keys" on the left navigation bar.



6. Press the "Add deploy key" button.



7. Two text boxes will appear on the screen. Fill in a title and the contents of your `deploy_key.pub` file. Make sure you use the `deploy_key.pub` file, not the private key stored in `deploy_key`. Do not allow write access.



8. Press the "Add Key" button. Now you're all set to clone this forked repository's code onto your production server.

## Pulling Project Code to the Server

Git is not yet installed on the server so the first step will be to get the system package dependency. Then we can pull down the application code from GitHub with our new deploy key.

1. Install the `git-core` package via apt on our remote machine.

```
sudo apt-get install git-core
```

- Clone the remote Git repository so we have a copy of it on the server. If the repository is private the deploy key will grant you access. Be sure to use your username in the below command with the fork we created earlier.

```
ssh-agent bash -c 'ssh-add /home/deployer/deploy_key/deploy_key; git clone git@github.com:makaimc/cyoa.git'
```

- You'll see a prompt asking if the authenticity of the host `github.com` is valid based on the key fingerprint. Enter 'yes' to continue.

- If you run into the following error, you need to make sure your newly created public key is associated with your GitHub account.

```
Permission denied (publickey).  
fatal: Could not read from remote repository.
```

```
Please make sure you have the correct access rights  
and the repository exists.
```

- If the clone works text similar to the following output will appear.

```
Cloning into 'cyoa'...  
remote: Counting objects: 224, done.  
remote: Total 224 (delta 0), reused 0 (delta 0), pack-reused 224  
Receiving objects: 100% (224/224), 535.18 KiB | 0 bytes/s, done.  
Resolving deltas: 100% (113/113), done.  
Checking connectivity... done.
```

- Now the repository is stored on the production server.

- If we want to update the code by pulling the latest from the remote server we can run the following command.

```
ssh-agent bash -c 'ssh-add /home/deployer/deploy_key/deploy_key; git pull srcin master'
```

That's all the manual work we need to do in this chapter. If you're ready to automate these steps within our Ansible playbook then continue reading below, otherwise we can begin setting up our PostgreSQL and Redis data stores in the next chapter.

## Automating Source Control

Let's automate as many of the above steps as possible. Note however that you'll have to manually create the deploy key and add it to GitHub just as we did in the above steps.

1. Update `prod/group_vars/all` to include the following new deployment variables at the end of the file.

```
# Chapter 5: Source Control
## this is the local directory with the deploy_key and deploy_key.pub
## files. do not include a trailing slash
local_deploy_key_dir: ~/fsp-deployment-guide/deploy_key
code_repository: ssh://git@github.com/makaimc/choose-your-own-adventure-presentations.git
```

2. Create a new file `prod/roles/common/tasks/git.yml` with the following contents to automate source control installation, copy the deploy key and clone the latest web app code.

```
###  
# Clones or pulls the repo from a public or private Git repository  
##  
- name: ensure Git is installed via the system package  
  apt: name=git-core state=present update_cache=yes  
  sudo: yes  
  
- name: create deploy key directory if it does not exist  
  file: path=/home/{{ deploy_user }}/deploy_key state=directory  
  
- name: ensure deploy key is on remote server  
  copy: src={{ local_deploy_key_dir }}/deploy_key
```

```
dest=/home/{{ deploy_user }}/deploy_key/deploy_key  
mode=0600 owner={{ deploy_user }} group={{ deploy_group }}  
  
- name: clone or pull latest web app code  
git: repo={{ code_repository }} dest={{ app_dir }}  
key_file=/home/{{ deploy_user }}/deploy_key/deploy_key  
accept_hostkey=yes
```

3. Update `prod/roles/common/tasks/main.yml` with the following line at the end of the file.

```
- include: git.yml
```

4. We can again run our Ansible playbook again with our shell script, this time to install Git, copy over our deploy key and clone the remote repository onto our server.

```
./deploy_prod.sh
```

Our Ansible playbook can now clone our Git repository using a separate deploy key. However, we don't yet have all the pieces in place that are necessary for our application. We need to set up our databases on the server to handle data storage.

## Next: Databases

Next we're going to install the relational database and get it ready for our application to connect to it.

## Additional source control resources

Source control is a necessary tool for every software project. The following resources are fantastic for a better general understanding of how to use source control systems properly.

- [Staging Servers, Source Control & Deploy Workflows, And Other Stuff](#)  
[Nobody Teaches You](#) is a comprehensive overview by Patrick McKenzie of why you need source control.
- [A visual guide to version control](#) provides real-life examples for why version control is necessary in software development.
- [An introduction to version control](#) shows the basic concepts behind version control systems.
- [About version control](#) reviews the basics of distributed version control systems.

## Additional Git resources

Git is the source control system we're using throughout this book. The following links are solid resources to learn more about Git.

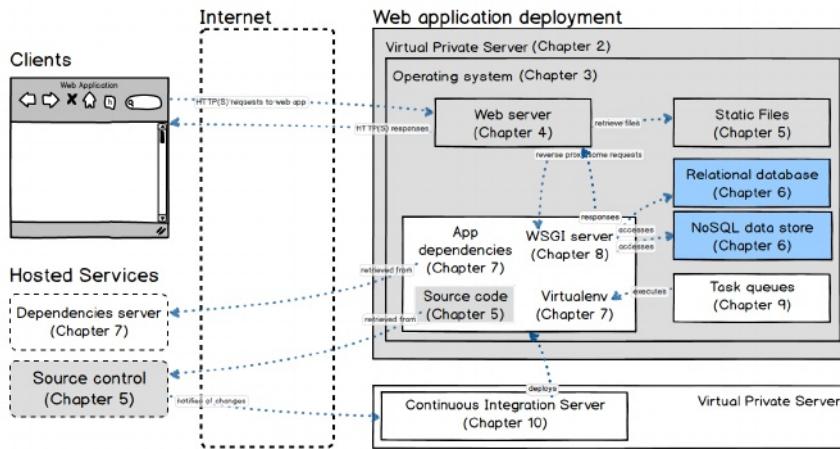
- [Pro Git](#) is a free open source book that walks through all aspects of using the version control system.
- [A Hacker's Guide to Git](#) covers the basics as well as more advanced Git commands while explaining each step along the way.
- [Git Workflows That Work](#) is a helpful post with diagrams to show how teams can create a Git workflow for their particular development process.

**Chapter 6**

# **Databases**

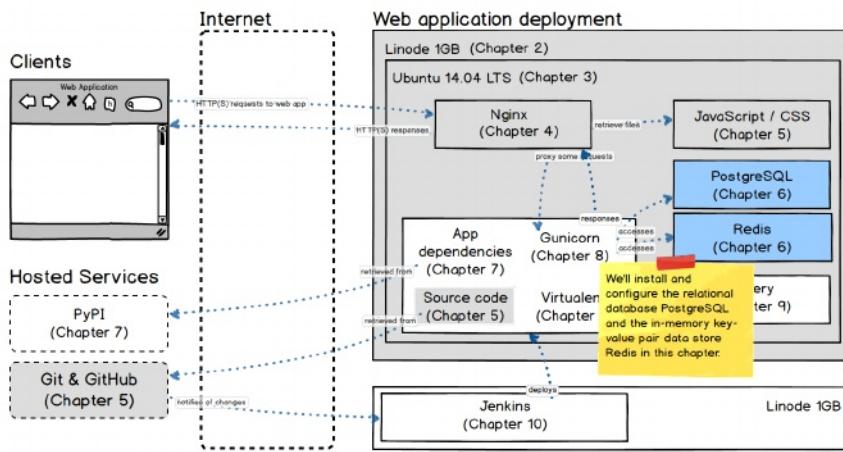
# Databases

A persistent database is an abstraction on top of an operating system's file system that makes it easier for applications to create, read, update, and delete persistent data.



Relational databases store data in structured tables that are persisted to the file system. Databases provide a mental framework for how the data should be saved and retrieved instead of having to figure out what to do with the data every time you build a new application.

In this chapter, we will set up two types of databases. The first is a traditional relational database called [PostgreSQL](#). The second database is a *NoSQL* data store. It is an in-memory key-value pair store called [Redis](#).



As shown in the above diagram, PostgreSQL will save our persistent data while Redis will store transient data.

## Databases in Our Deployment

We'll install and configure PostgreSQL and Redis for separate purposes. PostgreSQL will handle long-term persistent storage of users and presentations in the Choose Your Own Adventure Presentations application. Redis will keep transient data for the number of presentation votes that come in via text message voting.

Let's get to work on installing and configuring PostgreSQL.

# PostgreSQL

PostgreSQL is the recommended relational database for working with Python web applications. PostgreSQL's feature set, active development and stability contribute to its usage as the backend for millions of applications live on the Web today.

To work with a relational database using Python, you need to use a code library. In our case we will use [psycopg2](#) for PostgreSQL.

## Installing PostgreSQL

We need to install several new packages to get PostgreSQL, along with Python connection support, up and running.

- [postgresql](#) is the core database software
- [libpq-dev](#) provides development files so we can install the Python database connector known as psycopg2
- [postgresql-client-common](#) and [postgresql-client](#) are programs for connecting to the PostgreSQL database server

With explanations of those packages out of the way, let's get the database installed and configured.

1. PostgreSQL can be installed via the system packages. Run the following command to kick off the installation.

```
$ sudo apt-get install postgresql libpq-dev \
postgresql-client-common postgresql-client
```

- PostgreSQL is now installed but we need to create our database, set up a non-root user, set up our configuration and create our own tables.

- Switch over to the `postgres` user using the following command.

```
sudo su - postgres
```

- With the `postgres` user we can create the database that will hold our application's tables.

```
createdb cyoa
```

- Next let's set up a non-root database user to handle the table creation and maintenance.

```
createuser --superuser deployer
```

- We need to set a password for the deployer user. This password is separate from your superuser password so ensure you keep track of it so later we can configure the web application to connect to the database.

```
psql
ALTER USER deployer PASSWORD 'my_new_password';
```

- Exit out of the `psql` prompt by pressing CTRL-d.

- Log off the `postgres` user by again pressing CTRL-d.

- Now let's try out our PostgreSQL connection with the deployer user.

```
psql cyoa
```

- You should see a prompt like `cyoa=#`

- Enter the following command at the PostgreSQL prompt and we'll see no tables have been created yet for the `cyoa` database.

```
\dt
```

**No relations found.**

12. Exit the PostgreSQL prompt by pressing CTRL-d or entering `quit`
13. You may have noticed the error that PostgreSQL could not save history a file. We need to create a history file for Postgres commands. There just needs to be a blank file in the deployer's home directory. Run the touch command to create it.  
`touch ~/.psql_history`

Our PostgreSQL database is installed and we can access it from the deployer account, but there's nothing in it yet. We'll populate some initial data for our application in the next chapter after installing our application dependencies.

## NoSQL Data Stores

Relational databases store the vast majority of web application persistent data. However, there are several alternative classifications of storage representations.

1. Key-value pair
2. Document-oriented
3. Column-family table
4. Graph

These persistent data storage representations are commonly used to augment, rather than completely replace, relational databases.

In this section of the chapter, we'll install the Redis NoSQL data store to support our application.

## Redis

[Redis](#) is an open source in-memory key-value pair data store. Redis is often called "the Swiss Army Knife of web application development." It can be used for caching, queuing, and storing session data for faster access than a traditional relational database, among many other use cases. Key-value pair data stores are based on [hash](#) [map](#) data structures and when they're stored in memory they are much faster than accessing data from persistent storage.

## Installing Redis

It's possible to install Redis from source control but in this case it'll be easiest to install the Ubuntu 14.04 LTS package.

1. Install Redis from the apt repository.

```
sudo apt-get install redis-server
```

## Testing the Redis Installation

Both `redis-server` and the `redis-cli` command line interface are now installed.

If you already know how to use Redis you can skip this section if you already know how to use Redis as we're just going to run through a few basic commands.

1. We can test the installation using the following command.

```
redis-cli
```

2. We'll see something like the following command line interface.

```
127.0.0.1:6379>
```

3. Values can be retrieved from Redis via the `get` command followed by the key name. For example:

```
127.0.0.1:6379> get cyoa
(nil)
```

4. A value for the key 'cyoa' has not been set so the value returned is specified as `nil`. Let's set it now and get the new set value.

```
127.0.0.1:6379> set cyoa 1
OK
127.0.0.1:6379> get cyoa
"1"
```

5. Strings are the basic data type in Redis so we receive back our 1 as a String. However we can perform operations on the value that are more integer-like such as increment and decrement. We'll try that real quick before moving on with our deployment.

```
127.0.0.1:6379> incr cyoa
(integer) 2
127.0.0.1:6379> get cyoa
"2"
```

Redis understands the increment operation and increases the value to 2. When we retrieve the value through a get though Redis still returns the value back as the basic String data type.

Redis is operating with default settings which will suffice for our web application deployment. We can now continue our manual set up in the next chapter on application dependencies or automate this part of the deployment in the remainder of this chapter.

## Automating the PostgreSQL and Redis Installations

Let's augment our existing Ansible playbook to install and configure the PostgreSQL and Redis databases.

1. Append the following new deployment variables to the existing `prod/group_vars/all` file that will be used in this chapter to install and configure the databases.

```
# Chapter 6: Databases
db_url: postgresql://{{deploy_user}}:{{db_password}}@localhost/{{app_name}}
## make sure to change this password to what you want your
## database password to be!
db_password: fortheloveofgodpleaseuseagoodpassword
db_name: "{{ app_name }}"
db_user: "{{ deploy_user }}"
```

2. Add a new file `prod/roles/common/tasks/postgresql.yml` named with the following contents.

```
###  
# Installs and configures the PostgreSQL database  
#  
- name: ensure postgresql database packages are installed  
  apt: name={{item}}  
  sudo: yes  
  with_items:  
    - postgresql  
    - libpq-dev
```

```

- python-psycopg2
  - postgresql-client
  - postgresql-client-common

- name: ensure database is created
  postgresql_db: name={{ app_name }}
  sudo: yes
  sudo_user: postgres

- name: configure separate PostgreSQL user
  postgresql_user: db={{ app_name }} name={{ db_user }}
    password={{ db_password }} priv=ALL
    role_attr_flags=NOSUPERUSER
  sudo: yes
  sudo_user: postgres

```

3. Add one more new file called `prod/roles/common/tasks/redis.yml` with this YAML to install the `redis-server` package.

```

###  

# Installs and configures Redis  

##  

- name: ensure redis package is installed
  apt: name=redis-server state=present update_cache=yes
  sudo: yes

```

4. Update `prod/roles/common/tasks/main.yml` to include the following new highlighted lines.

```

###  

# configures the server and installs the web application  

##  

- include: ubuntu.yml

- include: nginx_ssl.yml
  when: deploy_ssl

- include: nginx_no_ssl.yml
  when: not deploy_ssl

- include: git.yml

```

```
- include: postgresql.yml  
- include: redis.yml  
  
- include: postgresql.yml  
- include: redis.yml
```

5. Make sure to again run our Ansible playbook again with our shell script to install and configure the databases on our server.

```
./deploy_prod.sh
```

## Next: Application Dependencies

In the next chapter we will handle the application dependencies with a virtualenv, `requirements.txt` file and PyPI.

### Additional Database Resources

Numerous resources for relational databases and NoSQL data stores are available to continue learning about the best ways to use these technologies. Here's a list of useful ones I've come across in the past few years.

- [DB-Engines](#) ranks the most popular database management systems.
- [DB Weekly](#) is a weekly roundup of general database articles and resources.
- [SQLAlchemy vs Other ORMs](#) provides a detailed comparison of SQLAlchemy against alternatives.
- [A different view](#) provides some perspective on the impedance mismatch between ORMs and traditional SQL queries.

- [Databases integration testing strategies](#) covers a difficult topic that comes up on every real world project.

## Additional PostgreSQL resources

PostgreSQL is the top of mind relational open source database today. Here are several resources that helped me better understand how to use it, especially with Python web applications.

- This post on [using PostgreSQL with Django or Flask](#) is a great quickstart guide for either framework.
- [PostgreSQL Weekly](#) is a weekly newsletter of PostgreSQL content from around the web.
- Braintree wrote about their experiences [scaling PostgreSQL](#). The post is an inside look at the evolution of Braintree's usage of the database.
- This post estimates the [costs of a PostgreSQL connection](#)
- There is no such thing as total security but this IBM article covers [hardening a PostgreSQL database](#)

## Additional NoSQL Resources

NoSQL is a tricky beast to figure out. In most application deployments, a relational database does the heavy lifting for persistent storage and NoSQL data stores are used for special cases, such as caching, indexing and search term retrieval.

- [CAP Theorem overview](#) is a ground up explanation of the principles behind data persistence mechanisms.

- [NoSQL Weekly](#) is a free curated email newsletter that aggregates articles, tutorials, and videos about non-relational data stores.
- [NoSQL comparison](#) is a large list of popular, BigTable-based, special purpose, and other data stores with attributes and the best use cases for each one.
- [What is a key/value store database?](#) is a Stack Overflow Q&A that straight on answers this subject.

## Additional Redis Resources

Here are some additional handy tutorials for installing and using Redis.

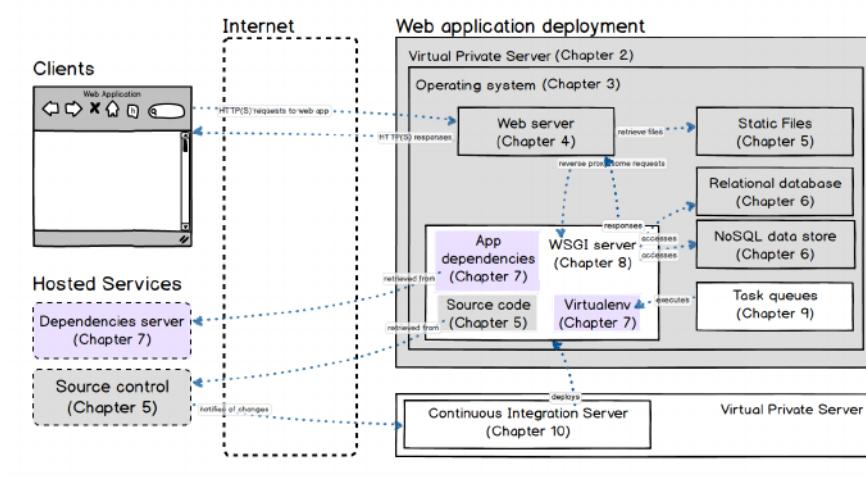
- [How To Install and Use Redis](#) provides a guide for getting up and running with the extremely useful in-memory data store.
- [Getting started with Redis and Python](#) is a walkthrough for installing and playing around with the basics of Redis.
- [Scaling Redis at Twitter](#) is a from the trenches story of Redis usage at the social media company.

**Chapter 7**

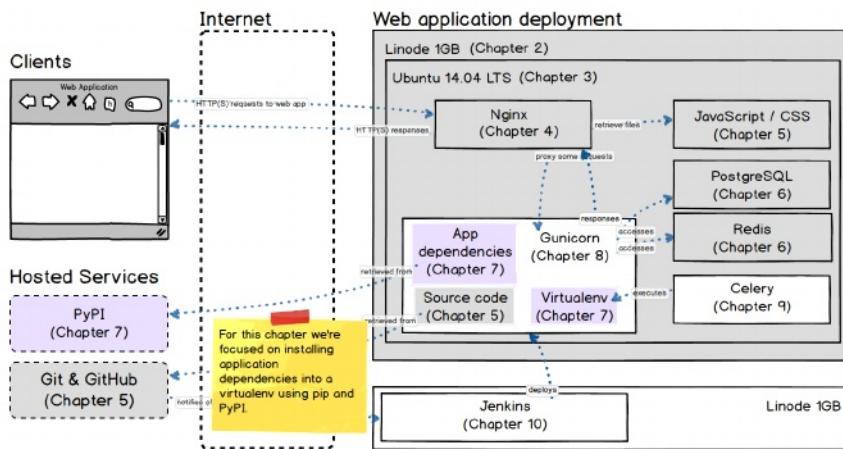
# **Application Dependencies**

# Application Dependencies

Application dependencies are the external libraries your project's code requires to execute your application. For example, our Choose Your Own Adventure Presentations web application relies on the [Flask microframework](#), so a reference to that library is stored in the project's `requirements.txt` file and will be installed as an application dependency.



In this chapter we will install our application dependencies via the `pip` command, which downloads the libraries from [PyPI](#) into a [virtualenv](#).



Let's take a look at how we'll use application dependencies and then dive into getting them set up for our production environment.

## Application Dependencies in Our Deployment

Python web applications are built upon the work done by thousands of open source programmers. Application dependencies include not only web frameworks but also libraries for scraping, parsing, processing, analyzing, visualizing and many other tasks. Python's ecosystem facilitates discovery, retrieval and installation so applications are easier for developers to create.

Our deployment will continue in this chapter by performing the following configuration:

1. Create a virtualenv for a fresh Python installation environment
2. Install the necessary application dependencies listed in the `requirements.txt` file by using the `pip` command to retrieve the libraries from [PyPI](#)
3. Populate environment variables so our program can access them

Before we jump into our manual steps let's take a quick look at why we need to isolate dependencies, how a `requirements.txt` file is used and explain dependency pegging.

## Isolating Dependencies

Dependencies are installed separately from system-level packages to prevent library version conflicts. The most common isolation method for Python application is using a [virtualenv](#). Each virtualenv is its own copy of the Python interpreter and dependencies in the `site-packages` directory. To use a virtualenv it must first be created with the `virtualenv` command and then activated.

The virtualenv stores dependencies in an isolated environment. The web application then relies only on that virtualenv instance to run.

## Storing Dependencies in requirements.txt

The Python convention for specifying application dependencies is a `requirements.txt` file. When you build a Python web application you should always include a `requirements.txt` file with your project's pegged dependencies. "Pegging"

dependencies means attaching a specific version number after the name of the dependency.

For example, in our Choose Your Own Adventure Presentations application the `requirements.txt` file with pegged dependencies looks like this:

```
Flask==0.10.1
Flask-Script==2.0.5
Flask-SocketIO==0.4.1
Flask-Login==0.2.11
Flask-SQLAlchemy==2.0
Flask-WTF==0.10.3

gunicorn==19.3.0
redis==2.10.3
twilio==3.7.2
psycopg2==2.5.4
celery==3.1.18
```

Pegged dependencies with precise version numbers or Git tags are important because otherwise the latest version of a dependency will be used. While it may sound good to always stay up to date, there's no telling if your application actually works with the latest versions of all dependencies. Developers should deliberately upgrade and test to make sure there were no backwards-incompatible modifications in newer dependency library versions.

## Creating Our Virtualenv

It's time to create our virtualenv for this project so we can run our Python code. Make sure you're logged into the remote production server with the deployer user to execute the following commands.

1. Create a directory to store virtualenvs.

```
mkdir /home/deployer/envs/
```

2. We'll call our new virtualenv **cyoa**, an acronym for "Choose Your Own Adventure".

```
virtualenv /home/deployer/envs/cyoa
```

3. Activate the virtualenv.

```
source /home/deployer/envs/cyoa/bin/activate
```

4. We should see a prefix to the shell like the following. That prefix helps us remember which virtualenv is currently active.

```
(cyoa)$
```

5. Now the virtualenv is activated for our shell. However, this activation is not automatic for every shell that is opened. Our virtualenv is only activated for the shell we explicitly activate it for.

6. We can also use other commands in the bin directory without activating the virtualenv first though. For example we could run the Python

interpreter within that virtualenv by running the following command.  
**/home/deployer/envs/cyoa/bin/python**

## Installing Dependencies

With our virtualenv activated we can install the web application's dependencies.

1. Install the necessary Choose Your Own Adventure Presentations project dependencies with **pip** into the **cyoa** virtualenv.

```
pip install -r /home/deployer/cyoa/requirements.txt
```

2. Copy the `template.env` file to a new file named `.env`.

```
cp template.env .env
```

3. Fill in the following necessary environment variables in the new `.env` file.

```
#!/bin/bash
export DEBUG=False
export SECRET_KEY='supersecretproductionkeyforapp'
export DATABASE_URL='postgres://username:password@localhost/cyoa'

# Redis settings
export REDIS_SERVER='localhost'
export REDIS_PORT='6379'
export REDIS_DB='1'

# Twilio settings
export TWILIO_ACCOUNT_SID=''
export TWILIO_AUTH_TOKEN=''
export TWILIO_NUMBER=''

# Celery
export CELERY_BROKER_URL='redis://localhost:6379/0'
export CELERY_RESULT_BACKEND='redis://localhost:6379/0'
```

4. Invoke the environment variables by running the shell script on the command line from the `cyoa` base directory.

```
(cyoa)$ . .env
```

We can test that the dependencies are properly installed and that the environment variables were invoked by syncing the web application with the database.

## Syncing the Database Schema

1. Our database is ready for table creation, so let's run a command from the `manage.py` file.

```
(cyoa)$ python manage.py syncdb
```

2. No output is likely a good sign that the tables created without a problem, but let's check PostgreSQL to be certain.
3. Connect to PostgreSQL with the deployer user.

```
psql cyoa
```

4. You'll see a prompt like `cyoa=#`
5. Enter the following command at the PostgreSQL prompt and we'll now see that tables have been created in the database for our application.

```
cyoa=# \dt
          List of relations
 Schema |      Name       | Type  |  Owner
-----+---------------+-----+-----
 public | choices      | table | deployer
 public | presentations | table | deployer
 public | wizards      | table | deployer
(3 rows)
```

## Populating Our Application's Initial Data

Most applications that are deployed need some initial data populated to run properly. In our Choose Your Own Adventure Presentations application we only need a Wizard user so we can log into the Wizard admin panel.

1. We've synced our database tables, but we need the user for our application. Run the following command to create a user with a username and password of your choice.

```
(cyoa)$ python manage.py create_wizard username password
```

Our database tables and new user are ready to go. If you want to finish off the manual application deployment and populate some of the initial database data, move on to the next chapter on WSGI server. For the remainder of this chapter we'll take a look at automating the above steps.

## Automating App Dependency Installation

It's time to make some further changes to our Ansible deployment playbook.

1. Update `prod/group_vars/all` with the following new highlighted lines at the end of the file.

```
# Chapter 3: Operating System (Ubuntu)
app_name: cyoa
deploy_user: deployer
deploy_group: deployers
## this is the local directory with the SSH keys and known_hosts
## file do not include a trailing slash
ssh_dir: ~/devel/py/fsp-deployment-guide/ssh_keys

# Chapter 4: Web Servers
## deploy_ssl true for HTTPS, false for only HTTP
deploy_ssl: false
wsgi_server_port: 8000
fqdn: cyoa.mattmakai.com
app_dir: /home/{{deploy_user}}/{{app_name}}
## local directory SSL certificates should be located here
ssl_certs_dir: ~/devel/py/fsp-deployment-guide/ssl_cert/
```

```

# Chapter 5: Source Control
local_deploy_key_dir: ~/fsp-deployment-guide/deploy_key
code_repository: ssh://git@github.com/makaimc/choose-your-own-adventure-presentations.git

# Chapter 6: Databases
db_url: postgresql://{{deploy_user}}:{{db_password}}@localhost/{{app_name}}
db_password: fortheloveofgodpleaseuseagoodpassword
db_name: "{{ app_name }}"
db_user: "{{ deploy_user }}"

# Chapter 7: Application Dependencies
venv_dir: "/home/{{ deploy_user }}/venvs/{{ app_name }}"
venv_python: "{{ venv_dir}}/bin/python"

wsgil_env_vars: {
    DEBUG: True,
    SECRET_KEY: 'jqwpifojqwoifjioqwjfoiqwj',
    DATABASE_URL: '{{ db_url }}',
    REDIS_SERVER: 'localhost',
    REDIS_PORT: 6379,
    REDIS_DB: 1,
    TWILIO_ACCOUNT_SID: 'ACcb4c69632458114d8bec0e9837d7dad8',
    TWILIO_AUTH_TOKEN: '61a236ab4cc21cf4915ca85a8c71ff2c',
    TWILIO_NUMBER: '+12027598445',
    CELERY_BROKER_URL: 'redis://localhost:6379/0',
    CELERY_RESULT_BACKEND: 'redis://localhost:6379/0',
}

```

```

}

## username you want to use to log into app's admin screen
app_admin_user: ''

## password you want to use to log into app's admin screen
app_admin_password: ''

```

2. Create a new file named `dependencies.yml` under `prod/roles/web/tasks`. Fill in the following lines in the new file.

```

###  

# Handles application dependencies and environment variables  

# Also syncs the application with the database schema  

##  

- name: create virtualenv directory if it does not already exist  

  file: path={{ venv_dir }} state=directory  
  

- name: install dependencies into a new virtualenv  

  pip: requirements={{ app_dir }}/requirements.txt  

    virtualenv={{ venv_dir }}  
  

- name: create .env file in base directory of the app from template  

  template: src=env.j2  

    dest={{ app_dir }}/.env  
  

- name: sync the database tables between Flask and PostgreSQL  

  shell: ". {{ app_dir }}/.env; {{ venv_python }} {{ app_dir }}/manage.py syncdb"  
  

- name: create the initial admin user in the database  

  shell: ". {{ app_dir }}/.env; {{ venv_python }} {{ app_dir }}/manage.py  

    create_wizard {{ app_admin_user }} {{ app_admin_password }}"

```

3. Create a new file named `prod/roles/web/templates/env.j2`. Ensure you created the file in the templates directory and not under tasks like the file we just created in the previous step. Fill in the following lines.

```

#!/bin/bash
{% for k, v in wsgi_env_vars.iteritems() %}
export {{ k }}="{{ v }}"

```

```
{% endfor %}
```

4. The above template loops through the WSGI environment variables set for `wsgi_env_vars` in the `group_vars/all` file and fills them into the `.env` file.
5. The `.env` file won't actually be used for running the application except if we need to SSH in manually and test the installation. Environment variables for our deployment are set in the Supervisor configuration file instead.
6. Update `prod/roles/common/tasks/main.yml` to include the following new highlighted lines.

```
###  
# configures the server and installs the web application  
##  
- include: ubuntu.yml  
  
- include: nginx_ssl.yml  
  when: deploy_ssl  
  
- include: nginx_no_ssl.yml  
  when: not deploy_ssl  
  
- include: git.yml  
- include: postgresql.yml  
- include: redis.yml  
  
-include: dependencies.yml
```

7. Run our Ansible playbook again with our shell script to install the application dependencies, set the environment variables and sync the database schema with the application's models.

```
./deploy_prod.sh
```

## Next: WSGI Server

In the next chapter we'll run the WSGI server, Gunicorn, which will execute our Python code and bring up our web application.

### Additional Application Dependency Resources

Application dependencies can be annoying to install on your production server when you just want to get your app up and running. This chapter walked you through the basic steps for making that happen and automating it in future deployments. Check out the following resources if you want more information to better understand what's happening and what advanced steps to take next.

- [Jon Chu](#) wrote a great introduction on [virtualenv and pip basics](#).
- [A non-magical introduction to virtualenv and pip](#) breaks down what problems these tools solve and how to use them.
- [Tools of the modern Python hacker](#) contains detailed explanations of virtualenv, Fabric, and pip.
- Occasionally arguments about using Python's dependency manager versus one of Linux's dependency managers comes up. This provides [one perspective on that debate](#).
- This Stack Overflow question details how to set up a [virtual environment for Python development](#).

## Python Package Resources

It's useful to browse through these lists in case you come across a library to solve a problem by reusing the code instead of writing it all yourself. A few of the best collections of Python libraries are:

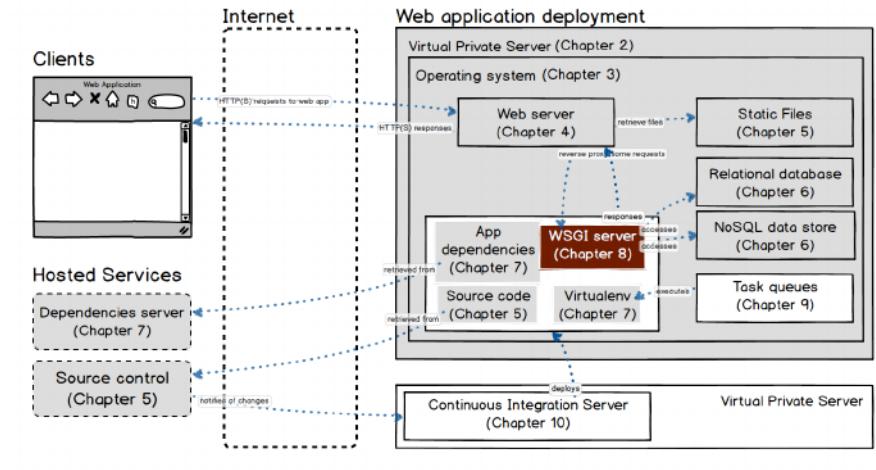
- [Python.org's useful modules](#) groups modules into categories.
- [GitHub Explore Trending repositories](#) shows the open source Python projects trending today, this week and this month.
- This list of [20 Python libraries you can't live without](#) is a wide-ranging collection from data analysis to testing tools.
- Wikipedia has an extensive [page dedicated to Python libraries](#) grouped by categories.

**Chapter 8**

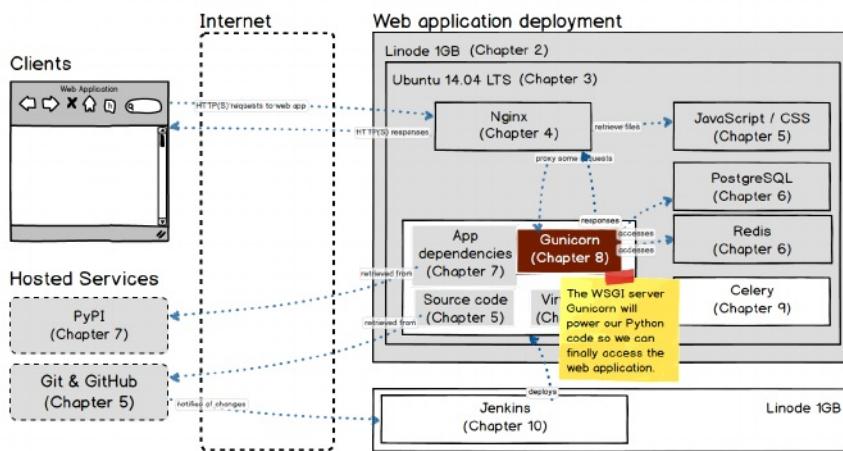
# WSGI Servers

# WSGI Servers

A [Web Server Gateway Interface](#) (WSGI) server runs a Python web application on the implements the web server side of the WSGI interface for running Python web applications.



The WSGI server implementation for this chapter is [Gunicorn](#), which is short for "Green Unicorn".



In this chapter we'll also populate initial data for our PostgreSQL database.

## What is WSGI?

Traditional web servers such as Apache and Nginx do not understand or have any way to run Python. In the late 1990s, a developer named Grisha Trubetskoy [came up with an Apache module called mod\\_python](#) to execute arbitrary Python code. For several years in the late 1990s and early 2000s, Apache configured with mod\_python ran most Python web applications.

However, mod\_python wasn't a standard specification. It was just an implementation that allowed Python code to run on a server. As mod\_python's development stalled and security vulnerabilities were discovered there was recognition by the community that a consistent way to execute Python code for web applications was needed.

Therefore the Python community came up with WSGI as a standard interface that modules and containers could implement. WSGI is now the accepted approach for running Python web applications.

If you're using a standard web framework that implements the WSGI application side such as Django, Flask or Bottle then your application can be deployed in the same way as the Choose Your Own Adventure Presentations application. Likewise, if you're using a standard WSGI container such as Green Unicorn, uWSGI, mod\_wsgi or gevent, you can get them running without worrying about how they implement the WSGI standard.

## WSGI Server Implementations

There is a comprehensive list of WSGI servers on the [WSGI Read the Docs](#) page. The following are WSGI servers based on community recommendations.

- [uWSGI](#) is gaining steam as a highly-performant WSGI server implementation.
- [mod\\_wsgi](#) is an Apache module implementing the WSGI specification.
- [CherryPy](#) is a pure Python web server that also functions as a WSGI server.
- [Green Unicorn](#) (Gunicorn) is a pre-fork worker model based server ported from the Ruby[Unicorn](#) project.

In this chapter, we're going to use the Green Unicorn WSGI server implementation to run our application.

# Configuring Green Unicorn

We're going to use Green Unicorn (gunicorn) as our WSGI server in this deployment. Fortunately gunicorn is specified as a dependency in our `requirements.txt` file so it's already installed in our virtualenv.

1. Activate the virtualenv if it is not already enabled.

```
source /home/deployer/envs/cyoa/bin/activate
```

2. Our virtualenv active prompt should show up on the command line.

```
(cyoa)$
```

3. Change into the `cyoa` project directory.

```
cd ~/cyoa
```

4. Activate our environment variables for the application.

```
.. .env
```

5. Start up gunicorn manually to ensure the web application is working.

```
(cyoa)$ gunicorn cyoa:app
```

Our application should be up and running now, so let's test it out. Go to the URL you've set up as your hostname. In my case, the URL to visit is <http://cyoa.mattmakai.com/>. If we see the following screen, our application is running properly.



No presentations are found yet since we haven't loaded any data into the application.

# Starting Gunicorn with Supervisor

We don't normally want to run the Gunicorn server ourselves by logging into the server. Instead, we'll use the Supervisor tool to start the WSGI server implementation for us.

Supervisor is not yet installed on the system so we need to obtain it via a system

package. Then we just need to create a configuration file and start the `supervisor` process once to get it up and running.

1. Install the `supervisor` package via apt on our remote machine.

```
sudo apt-get install supervisor
```

2. Create a new file within the `/etc/supervisor/conf.d` directory named `cyoa.conf` with the following contents. Remember that you will need to replace the environment values with your own application's values.

```
[program:cyoa]
environment=DEBUG=False,SECRET_KEY=mysupersecretkeyhere,REDIS_SERVER="localhost",REDIS_PORT=6379,REDIS_DB=0,TWILIO_ACCOUNT_SID="ACxxxxxx",TWILIO_AUTH_TOKEN="authtokenenvalue",TWILIO_NUMBER="+12025551234",DATABASE_URL="postgresql://deployer:databasepw@localhost/cyoa"
command=/home/deployer/envs/cyoa/bin/gunicorn cyoa:app
directory=/home/deployer/cyoa
user=deployer
autostart=true
autorestart=true
redirect_stderr=True
```

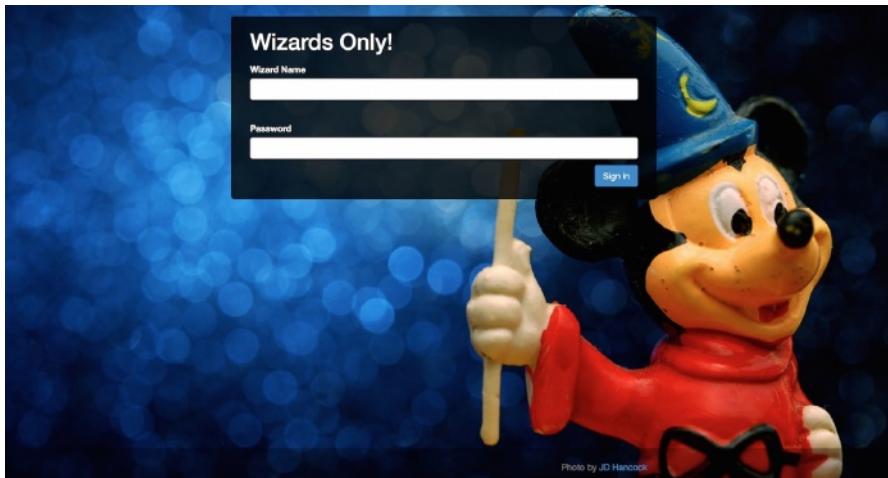
3. Save the file and exit.
4. Start Supervisor with the following command.

```
sudo service supervisor start
```

That'll do it for Supervisor, so let's test out our application.

# Our App is Live!

Our app should be up and running at the URL we specified now. Let's give it a test with the "Wizards Only" sign in page. Go to <http://cyoa.mattmakai.com/wizard> and you should see the following screen (replace the root URL with the server you set up to test your own application):



If that screen shows up, we're looking good! There aren't any users in the application at the moment though so we can't log in just yet. We will fix the no users issue in the next section.

## Automating Gunicorn Setup

1. Create a new file `prod/roles/common/tasks/wsgi.yml` that will have new YAML instructions for Ansible to execute.

```

###  

# Sets up and configures Supervisor which runs Green Unicorn  

##  

- name: ensure Supervisor is installed via the system package  

  apt: name=supervisor state=present update_cache=yes  

  sudo: yes  

- name: create Supervisor template for the WSGI app to run  

  template: src=supervisor_app.conf.j2  

    dest=/etc/supervisor/conf.d/{{ app_name }}.conf  

  sudo: yes  

- name: stop supervisor for app  

  command: service supervisor stop  

  sudo: yes  

- name: pause for restart to take effect  

  pause: seconds=2  

- name: start supervisor for app  

  command: service supervisor start  

  sudo: yes  

  notify:  

    - restart nginx

```

2. Add a new file `prod/roles/common/templates/supervisor_app.conf.j2` with the following content. **Important note:** there *must* be a blank line between the `environment=...` and `command=...` lines or the output from Ansible will merge onto one line. PDFs and EPUB formats often do not preserve the blank line in output so be sure to [check the tagged file on GitHub](#) for the correct YAML.

```

[program:{{ app_name }}]  

environment=% for k, v in wsgi_env_vars.iteritems() %}{% if not loop.first %},{%  

endif %}{{ k }}={{ v }}"%{ endfor %}  

  command={{ venv_dir }}/bin/gunicorn {{ app_name }}:app  

  directory={{ app_dir }}  

  user={{ deploy_user }}  

  autostart=true

```

```
autorestart=true  
redirect_stderr=True
```

3. Update the existing `prod/roles/common/tasks/main.yml` by appending the following highlighted line.

```
###  
# configures the server and installs the web application  
##  
- include: ubuntu.yml  
  
- include: nginx_ssl.yml  
  when: deploy_ssl  
  
- include: nginx_no_ssl.yml  
  when: not deploy_ssl  
  
- include: git.yml  
- include: postgresql.yml  
- include: redis.yml  
-include: dependencies.yml  
  
- include: wsgi.yml
```

4. As we've done in the last several chapters, we'll re-run our Ansible playbook with our shell script from the base directory of our deployment project.

```
./deploy_prod.sh
```

## Next: Task Queue

Our application is up and running, but we've got more configuration work to do. In order to handle asynchronous background tasks, we need to set up a task queue on

our server. We'll learn more about task queues and the Celery task queue implementation in the next chapter.

## Additional WSGI Resources

There are many resources available for both specifications and implementations of WSGI applications and servers. I recommend starting with the PEP 3333 standard as it's an accessible read and gives a good foundation for other materials you read about WSGI.

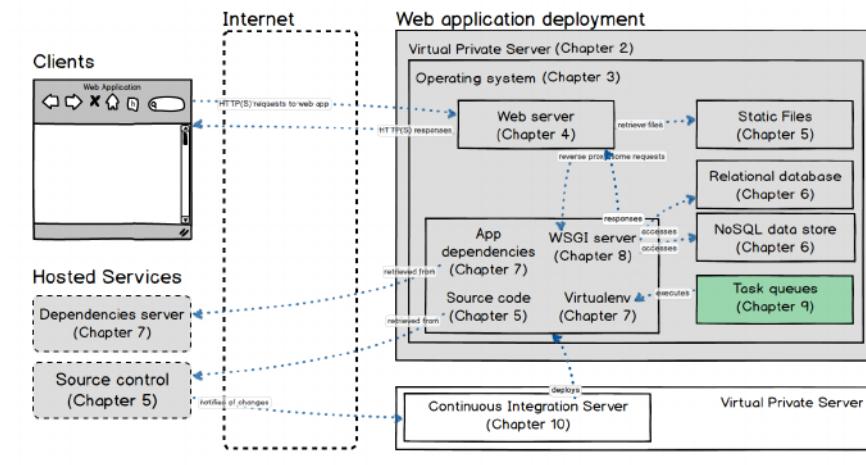
- The WSGI standard v1.0 is specified in [PEP 0333](#). As of September 2010, WSGI v1.0 is superseded by [PEP 3333](#), which defines the v1.0.1 WSGI standard.
- This [basics of WSGI](#) post contains a simple example of how a WSGI-compatible application works.
- The Python community made a long effort to [transition from mod\\_python](#) to the WSGI standard. That transition period is now complete and an implementation of WSGI should always be used instead mod\_python.
- A thorough and informative post for LAMP-stack hosting choices is presented in the [complete single server Django stack tutorial](#).
- Nicholas Piël wrote an interesting benchmark blog post of [Python WSGI servers](#). Note that the post is a few years old. Benchmarks should be considered for their specific tested scenarios and not quickly extrapolated as general "this server is definitely faster than this other server" results.

Chapter 9

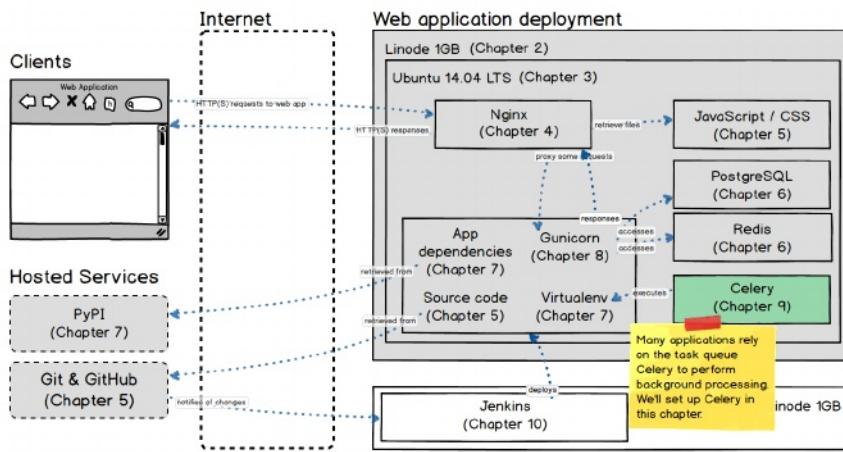
# Task Queues

# Task Queues

Task queues manage background work that must be executed outside the usual HTTP request-response cycle.



In this chapter we'll set up a particular task queue, [Celery](#), that is commonly used with Python web applications.



We'll connect Celery to Redis, which we previously set up in the databases chapter, for Celery to use as a message queue broker.

## What Are Task Queues For?

Tasks are handled asynchronously either because they are not initiated by an HTTP request or because they are long-running jobs that would dramatically reduce the performance of an HTTP response.

For example, a web application could poll the GitHub API every 10 minutes to collect the names of the top 100 starred repositories. A task queue would handle invoking code to call the GitHub API, process the results and store them in a persistent database for later use.

Another example is when a database query would take too long during the HTTP request-response cycle. The query could be performed in the background on a fixed interval with the results stored in the database. When an HTTP request comes in that needs those results a query would simply fetch the pre-calculated result instead of re-executing the longer query.

Other types of jobs for task queues include

- spreading out large numbers of independent database inserts over time instead of inserting everything at once
- aggregating collected data values on a fixed interval, such as every 15 minutes
- scheduling periodic jobs such as batch processes

## Task Queue Implementations

The defacto standard Python task queue is Celery. The other task queue projects that arise tend to come from the perspective that Celery is overly complicated for simple use cases. My recommendation is to put the effort into Celery's reasonable learning curve as it is worth the time it takes to understand how to use the project.

- The [Celery](#) distributed task queue is the most commonly used Python library for handling asynchronous tasks and scheduling.
- The [RQ \(Redis Queue\)](#) is a simple Python library for queueing jobs and processing them in the background with workers. RQ is backed by Redis and is designed to have a low barrier to entry. The [intro post](#) contains information on design decisions and how to use RQ.

- [Taskmaster](#)is a lightweight simple distributed queue for handling large volumes of one-off tasks.

Let's set up Celery manually now. As with previous chapter, later we will automate the installation with additions to our Ansible playbook.

## Configuring Celery

To set up our task queue Celery, we need to create two new Supervisor configuration files. Supervisor will run both the `celery` command, which handles the queue, and `celerybeat` which runs periodic tasks.

1. Log into your production server.
2. We're going to create two new templates for Supervisor: `celery` to process tasks and `celerybeat` to run periodic tasks.
3. Write a new file `/etc/supervisor/conf.d/supervisor_celery.conf` for Celery. You'll need to use the `sudo` command to write in that directory. For alpha values such as "localhost", you'll need to use double quotes around the value. For numeric values, quotes are not necessary but you can include them anyway.

```
[program:celery]
environment=environment=DEBUG=False,SECRET_KEY=mysupersecretkeyhere,REDIS_SERVER="lo
calhost",REDIS_PORT=6379,REDIS_DB=0,TWILIO_ACCOUNT_SID="ACxxxxxx",TWILIO_AUTH_TOKEN
="authokenvalue",TWILIO_NUMBER="+12025551234",DATABASE_URL="postgresql://
deployer:databasepw@localhost/cyoa",CELERY_RESULT_BACKEND="redis://localhost:
6379/0",CELERY_BROKER_URL="redis://localhost:6379/0"

command=/home/deployer/envs/cyoa/bin/celery worker -l info -P gevent -A cyoa.celery
directory=/home/deployer/cyoa
user=deployer
```

```
autostart=true  
autorestart=true  
redirect_stderr=True
```

4. Create another new file `/etc/supervisor/conf.d/supervisor_celerybeat.conf` for Celerybeat. You'll need to use the `sudo` command to write in that directory. Again, make sure to replace the values in the "environment" line with your own values.

```
[program:celerybeat]  
environment=environment=DEBUG=False,SECRET_KEY=mysupersecretpassword,REDIS_SERVER="localhost",REDIS_PORT=6379,REDIS_DB=0,TWILIO_ACCOUNT_SID="ACxxxxxx",TWILIO_AUTH_TOKEN="authTokenValue",TWILIO_NUMBER="+12025551234",DATABASE_URL="postgresql://deployer:databasepw@localhost/cyoa",CELERY_RESULT_BACKEND="redis://localhost:6379/0",CELERY_BROKER_URL="redis://localhost:6379/0"  
  
command=/home/deployer/envs/cyoa/bin/celery beat -A cyoa.celery  
directory=/home/deployer/cyoa  
user=deployer  
autostart=true  
autorestart=true  
redirect_stderr=True
```

5. Stop Supervisor.

```
sudo service supervisor stop
```

6. Pause just a moment to make sure Supervisor is completely stopped. Start Supervisor so the new configuration files take effect.

```
sudo service supervisor start
```

Supervisor is now up and running again with both the `celery` and `celerybeat` programs running.

# Automating Celery Installation

We can add a few files to our Ansible playbook to automate the task queue configuration.

1. Append the following new highlighted variables so that your `prod/group_vars/all` file looks similar to the following, but with your specific

values instead of the defaults.

```
# Chapter 3: Operating System (Ubuntu)
app_name: cyoa
deploy_user: deployer
deploy_group: deployers
## this is the local directory with the SSH keys and known_hosts
## file do not include a trailing slash
ssh_dir: ~/fsp-deployment-guide/ssh_keys

# Chapter 4: Web Servers
## deploy_ssl true for HTTPS, false for only HTTP
deploy_ssl: false
wsgi_server_port: 8000
fqdn: cyoa.mattmakai.com
app_dir: /home/{{deploy_user}}/{{app_name}}
## local directory SSL certificates should be located here
ssl_certs_dir: ~/fsp-deployment-guide/ssl_cert/

# Chapter 5: Source Control
local_deploy_key_dir: ~/fsp-deployment-guide/deploy_key
code_repository: ssh://git@github.com/makaimc/choose-your-own-adventure-presentations.git

# Chapter 6: Databases
db_url: postgresql://{{deploy_user}}:{{db_password}}@localhost/{{app_name}}
## make sure to change this password to what you want your
## database password to be!
db_password: fortheloveofgodpleaseuseagoodpassword
db_name: "{{ app_name }}"
```

```

db_user: "{{ deploy_user }}"

# Chapter 7: Application Dependencies
venv_dir: "/home/{{ deploy_user }}/venvs/{{ app_name }}"
venv_python: "{{ venv_dir}}/bin/python"
flask_env_vars: {
    ## in production you'll likely want to set this to False
    DEBUG: True,
    ## make sure to set your own unique long secret key for your app
    SECRET_KEY: 'super secret key make sure to change this',
    DATABASE_URL: '{{ db_url }}',
    REDIS_SERVER: 'localhost',
    REDIS_PORT: 6379,
    REDIS_DB: 1,
    ## these next two environment variables are found in the
    ## Twilio account dashboard on twilio.com
    TWILIO_ACCOUNT_SID: 'found on the twilio dashboard',
    TWILIO_AUTH_TOKEN: 'also found on the twilio dashboard',
    ## grab a number on Twilio for the application and Ansible
    TWILIO_NUMBER: '+15551234567',
    CELERY_BROKER_URL: 'redis://localhost:6379/0',
    CELERY_RESULT_BACKEND: 'redis://localhost:6379/0',
}
## username you want to use to log into app's admin screen
app_admin_user: ''
## password you want to use to log into app's admin screen
app_admin_password: ''

# Chapter 8: WSGI Servers

## no new variables necessary for this chapter

# Chapter 9: Task queues
deployment_alert: true
## your phone number that should receive a deployment complete
## text message. example format: "+19735551234"

```

```
    alert_number: ""
```

2. Create a new file `prod/roles/common/tasks/celery.yml` that will configure both the `celery` and `celerybeat` programs that are necessary for asynchronous and periodic tasks in our application.

```
###  
# Sets up and configures Supervisor for Celery and Celerybeat  
##  
- name: create Supervisor template for Celery to run  
  template: src=supervisor_celery.conf.j2  
            dest=/etc/supervisor/conf.d/{{ app_name }}_celery.conf  
  sudo: yes  
  
- name: create Supervisor template for Celerybeat to run  
  template: src=supervisor_celerybeat.conf.j2  
            dest=/etc/supervisor/conf.d/{{ app_name }}_celerybeat.conf  
  sudo: yes  
  
- name: stop supervisor for celery  
  command: service supervisor stop  
  sudo: yes  
  
- name: pause for restart to take effect  
  pause: seconds=2  
  
- name: start supervisor for celery  
  command: service supervisor start  
  sudo: yes
```

3. Create two new templates for Supervisor, one each for `celery` and `celerybeat`
4. Write a new file `prod/roles/common/templates/supervisor_celery.conf.j2` for Celery.  
**Important note:** there *must* be a blank line between the `environment=...` and `command=` lines or the output from Ansible will merge onto one line. PDFs

and EPUB formats often do not preserve the blank line in output so be sure to [check the tagged file on GitHub](#) for the correct YAML.

```
[program:celery]
environment=% for k, v in wsgi_env_vars.iteritems() %}{% if not loop.first %},{%
endif %}{{ k }}="{{ v }}"{% endfor %

command={{ venv_dir }}/bin/celery worker -l info -P gevent -A {{ app_name }}.celery
directory={{ app_dir }}
user={{ deploy_user }}
autostart=true
autorestart=true
redirect_stderr=True
```

5. Write another new file `prod/roles/common/templates/supervisor_celerybeat.conf.j2` for Celerybeat. Again, one more **important note**: there *must* be a blank line between the `environment=..` and `command=` lines or the output from Ansible will merge onto one line. PDFs and EPUB formats often do not preserve the blank line in output so be sure to [check the tagged file on GitHub](#) for the correct YAML.

```
[program:celerybeat]
environment=% for k, v in wsgi_env_vars.iteritems() %}{% if not loop.first %},{%
endif %}{{ k }}="{{ v }}"{% endfor %

command={{ venv_dir }}/bin/celery beat -A {{ app_name }}.celery
directory={{ app_dir }}
user={{ deploy_user }}
autostart=true
autorestart=true
redirect_stderr=True
```

6. Update `prod/roles/common/tasks/main.yml` to include the new `celery.yml` file at the end and send a text message alert once the deployment is completed.

```
- include: celery.yml

- name: alert that deployment is complete
```

```
twilio:  
  msg: "Deployment complete!"  
  account_sid: "{{ wsgi_env_vars.TWILIO_ACCOUNT_SID }}"  
  auth_token: "{{ wsgi_env_vars.TWILIO_AUTH_TOKEN }}"  
  from_number: "{{ wsgi_env_vars.TWILIO_NUMBER }}"  
  to_number: "{{ alert_number }}"  
  delegate_to: localhost  
  when: deployment_alert
```

7. Re-run our Ansible playbook with our shell script from the base directory of our deployment project. This time we won't have to wait in front of the computer screen for the deployment to finish. If you've set up the `alert_number` variable to your cell phone, you'll receive a text message stating that the deployment finished when it's done with all the steps.

```
./deploy_prod.sh
```

## Next: Continuous Integration

We've got our Celery task queue up and running with background jobs. Next up we will use our Ansible playbooks to automate the entire deployment to production whenever code is committed and pushed to the master branch of our repository on GitHub.

### Additional Task Queue Resources

Task queues are critical for creating performant web applications by shifting workloads into the background instead of trying to handle everything during the HTTP request-response cycle. Here are more resources for learning about task queue concepts.

- [Queues.io](#) is a collection of task queue systems with short summaries for each one. The task queues are not all compatible with Python but ones that work with it are tagged with the "Python" keyword.
- [Why Task Queues](#) is a presentation for what task queues are and why they are needed.
- [Asynchronous Processing in Web Applications: Part One](#) and [Part Two](#) are great reads for understanding the difference between a task queue and why you shouldn't use your database as one.

## Additional Celery Resources

Celery is a fantastic open source implementation of task queues. Here are few further resources on when and when not to use Celery in your projects.

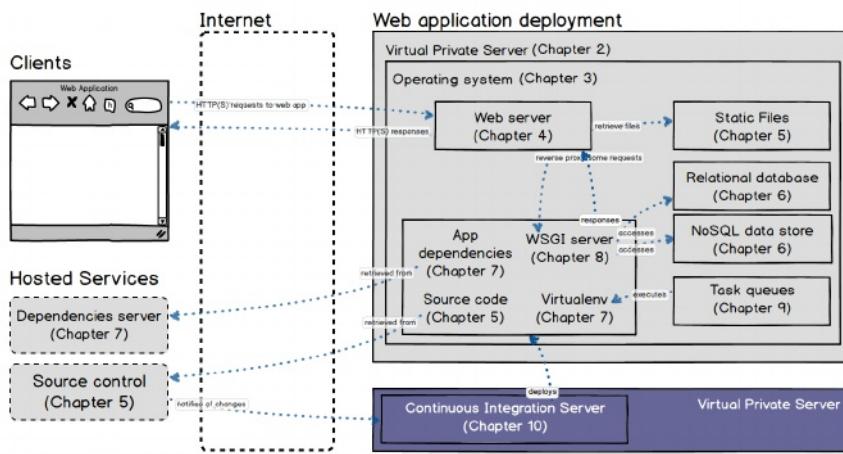
- [Getting Started Scheduling Tasks with Celery](#) is a detailed walkthrough for setting up Celery with the Django wen far,ewprl.
- [Celery - Best Practices](#) explains things you should not do with Celery and shows some underused features for making task queues easier to work with.
- [Distributing work without Celery](#) provides a scenario in which Celery and RabbitMQ are not the right tool for scheduling asynchronous jobs.
- [Introducing Celery for Python+Django](#) provides an introduction to the Celery task queue.

Chapter 10

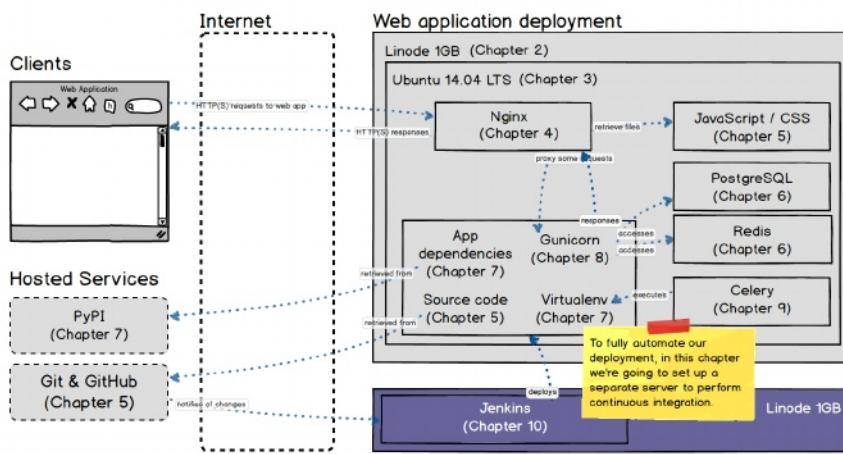
# Continuous Integration

# Continuous Integration

Continuous integration (CI) automates building, testing and deploying applications.



In this chapter, we'll set up Jenkins as our continuous integration server.



Jenkins will listen for HTTP POST requests from GitHub alerting it to when new code is pushed to our repository.

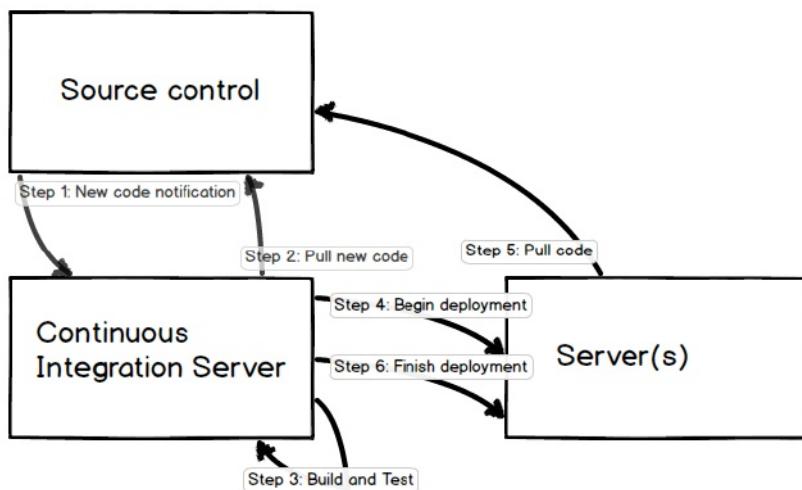
## How Will We Use Continuous Integration?

When CI is set up well it can dramatically reduce deployment times by eliminating manual steps and ensure code does not have bugs that are being checked by automated tests. Source code changes as a project evolves. CI combined with unit and integration tests check that code modifications do not break existing tests ensure the software works as intended.

# Setting Up Continuous Integration

For our project we're going to set up [Jenkins](#) to pull code from GitHub, build and test it then deploy it to our server. Jenkins is widely used and open source. However, there is a list of other open source and hosted continuous integration servers listed at the end of the chapter in case you want to try a different CI server out.

The continuous integration and deployment environment we set up will look like the following diagram.



When we push changes to our remote source control repository on GitHub, an HTTP POST request will be sent to our Jenkins server. That POST request informs Jenkins that a new build is required. Jenkins will pull the latest code from GitHub, make sure it's ready then deploy it to our production server using the Ansible playbook we've written throughout the book.

In a larger environment continuous integration would typically deploy the newest code to a test server for some manual checks before going to production. However, with this application we're keeping our set up as simple as possible and just deploying it straight into production.

Let's get moving to set up our continuous integration environment!

## Provisioning a New Server

We need another virtual private server to host our continuous integration environment so we're going to create a new public-private key pair and lock down the server using our Fabric script from chapter 2.

1. On your local machine, create a new directory at the base directory of our deployment project named `jenkins_ci`

```
mkdir jenkins_ci  
cd jenkins_ci
```

2. Execute the following command on your local machine to create a new public-private key pair just for the continuous integration server.

```
# we're running ssh-keygen command to generate the key pair  
# -b argument specifies the number of bits, 2048 in our case  
# along with -t argument to specify the RSA algorithm  
ssh-keygen -t rsa -b 2048
```

3. When prompted for a file in which to save the keys do not use the default. Instead, enter the following directory and file into the prompt.

```
# we are saving the private key in our local jenkins_ci directory  
./jenkins_key
```

4. Press enter twice when prompted for a passphrase. We will not use a passphrase for the Jenkins server user's keys.
5. Now you have two new files: a private key `jenkins_key` and a public key `jenkins_key.pub`. As with our previous keys, the private key should never be shared with anyone as it will allow them to access your server. The public key can be shared with anyone and does not need to be kept secret.
6. Execute the following command to create a new file with the newly created key as an authorized key.

```
# the authorized_keys file will allow our server to grant SSH
# access to a connection using the private key
cp jenkins_key.pub authorized_keys
```

7. Obtain a new separate server on Linode. We provisioned our Linode production server all the way back in chapter 2 so you may want to review those steps if you run into trouble.
8. We'll secure the server with the `prod/fabfile.py` Fabric script from chapter 2.

Copy the `fabfile`. Modify the values in the script with the new IP address.  
`cp prod/fabfile.py jenkins_ci/fabfile.py`

9. Edit the IP address found on line 21 of `fabfile.py` to the new Linode server's IP address.

```
env.hosts = ['192.168.1.1']
```

10. Edit line 35 in `fabfile.py` to specify a different key pair to upload to the server.

```
env.ssh_key_dir = '~/fsp-deployment-guide/jenkins_ci'
```

11. Modify line 77 in `fabfile.py` so the `jenkins_key.pub` file is uploaded instead of `prod_key.pub`.

```
'/jenkins_key.pub' + env.ssh_key_dir + \
```

12. Run the copied Fabric file with the following command.

```
fab bootstrap
```

13. SSH into the continuous integration server with the new user. Again, make sure to replace the IP address seen here with the IP address of your new server.

```
ssh -i jenkins_key deployer@192.168.1.1
```

Our server is provisioned, locked down with a public-private key pair and we've logged in for the first time. Now we need to actually get Jenkins up and running on the server.

## Tweaking Our Ansible Automation

Before we install and configure Jenkins, we need to make a couple small tweaks to our Ansible playbook. Currently Ansible asks for the sudo password of the `deployer` user when we execute the playbook. However, our continuous integration server will not be able to respond to command prompts.

1. Modify the `prod/hosts` file with the `ansible_sudo_pass` argument after the IP address of the production server you're using to host your application.

```
192.168.1.1 ansible_sudo_pass="{{ deploy_user_password }}"
```

2. Add the following highlighted lines to the end of the `prod/group_vars/all` variables file.

```
# Chapter 3: Operating System (Ubuntu)
app_name: cyoa

deploy_user: deployer
```

```

deploy_group: deployers
## this is the local directory with the SSH keys and known_hosts
## file. do not include a trailing slash
ssh_dir: ~/fsp-deployment-guide/ssh_keys

# Chapter 4: Web Servers
## deploy_ssl true for HTTPS, false for only HTTP
deploy_ssl: false
wsgi_server_port: 8000
fqdn: cyoa.mattmakai.com
app_dir: /home/{{deploy_user}}/{{app_name}}
## local directory SSL certificates should be located here
## do not include a trailing slash
ssl_certs_dir: ~/fsp-deployment-guide/ssl_cert

# Chapter 5: Source Control
## directory where deploy_key was created and stored.
## do not include a trailing slash
local_deploy_key_dir: ~/fsp-deployment-guide/deploy_key
code_repository: ssh://git@github.com/makaimc/choose-your-own-adventure-presentations.git

# Chapter 6: Databases
db_url: postgresql://{{deploy_user}}:{{db_password}}@localhost/{{app_name}}
## make sure to change this password to what you want your
## database password to be!
db_password: fortheloveofgodpleaseuseagoodpassword
db_name: "{{ app_name }}"
db_user: "{{ deploy_user }}"

# Chapter 7: Application Dependencies
venv_dir: "/home/{{ deploy_user }}/venvs/{{ app_name }}"
venv_python: "{{ venv_dir}}/bin/python"
wsgi_env_vars: {
    ## in production you'll likely want to set this to False
    DEBUG: True,
    ## make sure to set your own unique long secret key for your app
    SECRET_KEY: 'super secret key make sure to change this',
    DATABASE_URL: '{{ db_url }}',
    REDIS_SERVER: 'localhost',
    REDIS_PORT: 6379,
    REDIS_DB: 1,
}

```

```

## these next two environment variables are found in the
## Twilio account dashboard on twilio.com
TWILIO_ACCOUNT_SID: 'found on the twilio dashboard',
TWILIO_AUTH_TOKEN: 'also found on the twilio dashboard',
## grab a number on Twilio for the application and Ansible
TWILIO_NUMBER: '+15551234567',
CELERY_BROKER_URL: 'redis://localhost:6379/0',
CELERY_RESULT_BACKEND: 'redis://localhost:6379/0',
}

## username you want to use to log into app's admin screen
app_admin_user: ''
## password you want to use to log into app's admin screen
app_admin_password: ''

# Chapter 8: WSGI Servers
## no new variables necessary for this chapter

# Chapter 9: Task queues
deployment_alert: true
## your phone number that should receive a deployment complete
## text message. example format: "+19735551234"
alert_number: ""

# Chapter 10: Continuous Integration

## this grabs the sudo password for the deployer user
## from an environment variable called REMOTE_DEPLOYER_PASSWORD

deploy_user_password: "{{ lookup('env', 'REMOTE_DEPLOYER_PASSWORD') }}"

```

3. In the above Ansible variables file, we're now pulling the password from an environment variable instead of keeping it in the file itself. This is a much better security practice than keeping plaintext passwords in the variables file itself.
4. Test out the script locally by setting an environment variable with the password to the remote deployer user.

```
export REMOTE_DEPLOYER_PASSWORD='my super secret password'
```

5. If you're using the `deploy_prod.sh` shell script, remove the `-K` argument so the ansible-playbook command looks like the following:

```
#!/bin/bash
ansible-playbook ./prod/deploy.yml --private-key=\
./ssh_keys/prod_key -u deployer -i ./prod/hosts
```

With those minor tweaks in place, we can now continue installing and configuring our Jenkins CI server.

## Jenkins Package Installation

Next let's install the Jenkins package and handle the initial configuration.

1. Now that we're logged in make sure the necessary packages are installed on this server.

```
sudo apt-get update
sudo apt-get install fail2ban git-core python-virtualenv python-dev
```

2. We need to install Jenkins, but the `jenkins` package is in a third party repository.

```
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | \
sudo apt-key add -
```

3. Add the new package repository into our `source.list` file with the following command.

```
sudo add-apt-repository "deb http://pkg.jenkins-ci.org/debian binary/"
```

4. We're ready to update the available packages list and install our `jenkins` package.

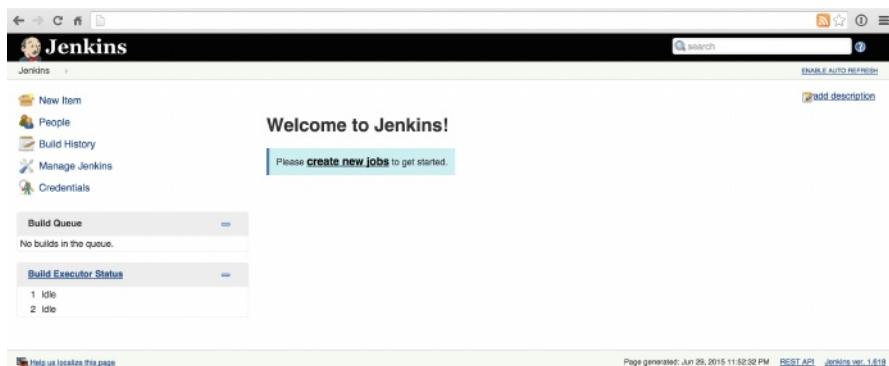
```
sudo apt-get update
sudo apt-get install jenkins
```

5. Once the package installations finish start up Jenkins as a system service.

```
sudo service jenkins start
```

6. Head to your server's IP address plus the port number 8080 in the browser. For example, if your Linode's IP address is 192.168.1.1, go to <http://192.168.1.1:8080/>.

You should see a screen like this one.



There's a massive problem with this screen though - if you can see it, so can anyone else! We need to secure the installation so only we can use it.

# Securing Jenkins with Authentication

Jenkins is open to the whole world until we secure it. We'll handle those steps now with a combination of command-line configuration and modifying settings in the Jenkins user interface.

1. Add the GitHub plugin.

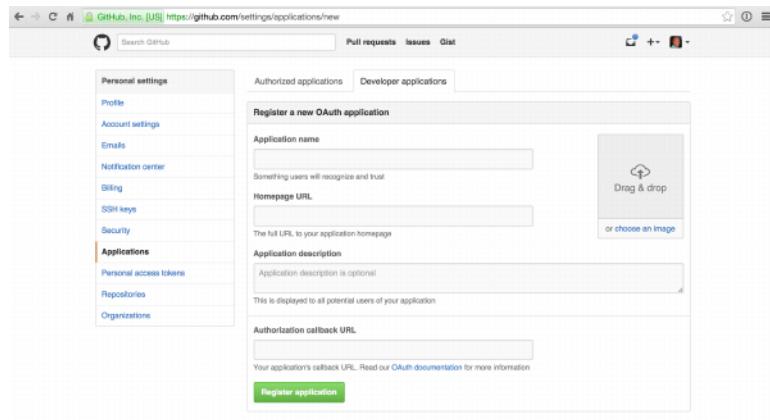
<input type="checkbox"/>	<a href="#">Google Calendar Plugin</a>	0.4
<input type="checkbox"/>	This plugin publishes build records over to <a href="#">Google Calendar</a>	
<input type="checkbox"/>	<a href="#">Gerrit Plugin</a>	0.7
<input type="checkbox"/>	This plugin integrates <a href="#">Gerrit Code Review</a> to Jenkins.	
<input checked="" type="checkbox"/>	<a href="#">GitHub Plugin</a>	1.11.3
	This plugin integrates Jenkins with <a href="#">Github</a> projects.	
<input type="checkbox"/>	<a href="#">GitHub SQS Plugin</a>	1.5
	This plugin integrates Jenkins with <a href="#">Github</a> projects via <a href="#">Amazon's Simple Queue Service</a>	
<input type="checkbox"/>	<a href="#">Google APIs Client Library</a>	2.0-1.20.0
	This plugin provides the <a href="#">Google APIs Client Library for Java</a> to other plugins.	

2. Add the GitHub OAuth plugin.

<input type="checkbox"/>	<a href="#">Crowd 2 Plugin</a>	1.8
<input type="checkbox"/>	This plugin enables use of <a href="#">Atlassian Crowd</a> as an authentication source.	
<input type="checkbox"/>	<a href="#">Extended Read Permission Plugin</a>	1.0
<input type="checkbox"/>	This plugin enables the Extended Read permission in Hudson 1.324 and newer.	
<input type="checkbox"/>	<a href="#">Favorite Plugin</a>	1.16
	This plugin allows you to mark a job a favorite.	
<input checked="" type="checkbox"/>	<a href="#">Github OAuth Plugin</a>	0.20
	Authentication of users is delegated to Github using the OAuth protocol. Authorization is based on the characteristics of the users Github user data that is retrieved through the Github API (effectively as the authenticated user).	
<input type="checkbox"/>	<a href="#">Google Login Plugin</a>	1.1
	This is a Jenkins plugin which lets you login to Jenkins with your Google account. Also allows you to restrict access to accounts in a given Google Apps domain.	
<input type="checkbox"/>	<a href="#">Gravatar plugin</a>	2.1
	This plugin shows <a href="#">Gravatar</a> avatars instead of the generic user image.	
	<a href="#">Hudson Personal View</a>	

3. Install the plugins and click for Jenkins to restart.

4. Go to <https://github.com/settings/applications/new> and you'll see the following screen.



5. Set up a new application for Jenkins on the GitHub screen. Fill in <http://192.168.1.1:8080/securityRealm/finishLogin> under the Authorization callback URL, where **192.168.1.1** is replaced with your Jenkins server IP address.

Authorized applications      Developer applications

**Register a new OAuth application**

**Application name**  
Jenkins CI - CYOA  
Something users will recognize and trust

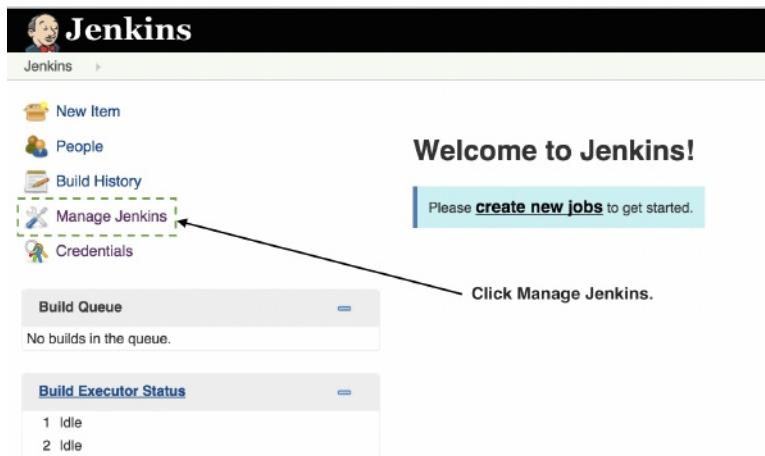
**Homepage URL**  
<https://github.com/makaimc/choose-your-own-adventure-presentations>  
The full URL to your application homepage

**Application description**  
Choose Your Own Adventure Presentations application.  
This is displayed to all potential users of your application

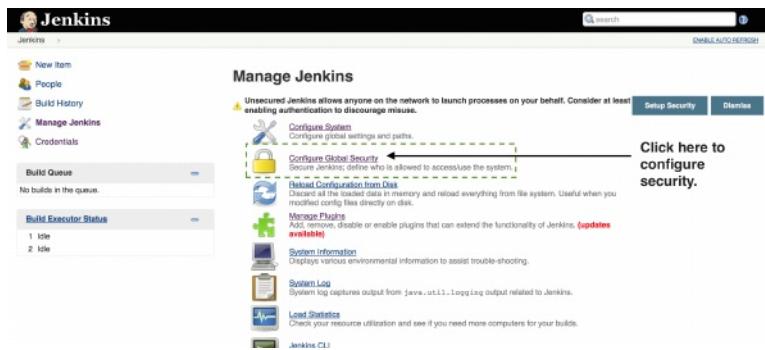
**Authorization callback URL**  
<http://192.168.1.1:8080/securityRealm/finishLogin>  
Your application's callback URL. Read our [OAuth documentation](#) for more information

**Register application**

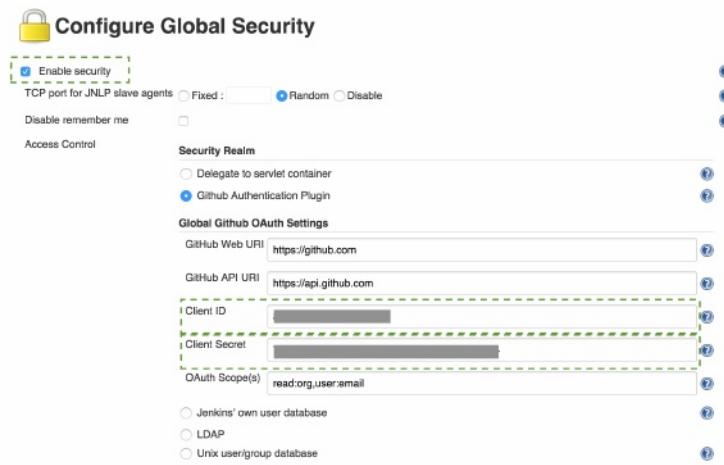
6. Move back over to the Jenkins web application now that it's restarted with the new plugins.
7. Click "Manage Jenkins".



8. Click "Configure Global Security".



9. Check "Enable Security", then enter the credentials GitHub generated for you when you created an application for Jenkins CI.



10. Select "Matrix-based Security", enter your GitHub username, click Add, select all permissions for your username but none for Anonymous, then click Apply.

Select Matrix-based security

Matrix-based security

Anonymous

User/group to add: makaimc

Add

Enter your GitHub username here then click the "Add" button.

Do not hit Apply until you've added your user and enabled all roles for that user.

	Slave	Job	Run	View	SCM
makaimc	<input type="checkbox"/> Connect <input checked="" type="checkbox"/> Create <input checked="" type="checkbox"/> Delete <input type="checkbox"/> Disconnect <input type="checkbox"/> Build <input type="checkbox"/> Cancel <input type="checkbox"/> Configure	<input type="checkbox"/> Create <input checked="" type="checkbox"/> Delete <input type="checkbox"/> Discover <input type="checkbox"/> Read <input type="checkbox"/> Workspace <input type="checkbox"/> Delete <input type="checkbox"/> Update <input type="checkbox"/> Configure	<input type="checkbox"/> Create <input checked="" type="checkbox"/> Delete <input type="checkbox"/> Discover <input type="checkbox"/> Read <input type="checkbox"/> Workspace <input type="checkbox"/> Delete <input type="checkbox"/> Update <input type="checkbox"/> Configure	<input type="checkbox"/> Create <input checked="" type="checkbox"/> Delete <input type="checkbox"/> Discover <input type="checkbox"/> Read <input type="checkbox"/> Workspace <input type="checkbox"/> Delete <input type="checkbox"/> Update <input type="checkbox"/> Configure	<input type="checkbox"/> Create <input checked="" type="checkbox"/> Delete <input type="checkbox"/> Discover <input type="checkbox"/> Read <input type="checkbox"/> Workspace <input type="checkbox"/> Delete <input type="checkbox"/> Update <input type="checkbox"/> Configure

Click this button to enable all roles for your authenticated account.

- Log in and out of Jenkins to confirm that a logged out user cannot access anything inside the application and is simply redirected to GitHub to determine if she should have access to this Jenkins CI user interface.

# Creating a CI Deploy Key

We need to create one more deploy key separate from the production server deploy key and the other private keys we use to log into the server. This deploy key will grant read-only access from the Jenkins server to the forked repository that we want to use the commands `git clone` and `git pull` to grab our code.

1. Log back into the Jenkins server via SSH. Again, make sure to replace the IP address seen here with the IP address of your new server.

```
ssh -i jenkins_key deployer@192.168.1.1
```

2. Change over to the `jenkins` user.

```
sudo su -s /bin/bash jenkins
```

3. Make sure you are in the `jenkins` home directory to execute the following commands.

```
cd ~
```

4. On the remote server create a new directory for our SSH keys. If the

directory already exists you can ignore the error message.

```
mkdir ~/.ssh
```

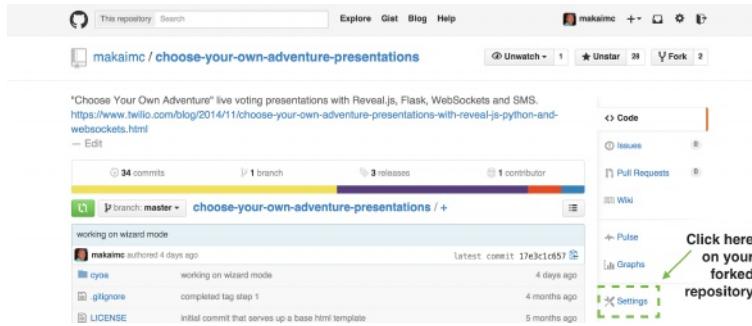
5. Execute the following command to create a new key pair.

```
ssh-keygen -t rsa -b 2048
```

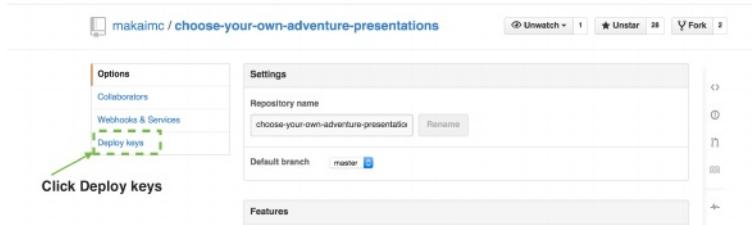
6. When prompted for a file in which to save the keys you can use the default values.

7. Press enter twice when prompted for a passphrase. We won't use a passphrase on the deploy key.

- Now you again have two new files: a private key `id_rsa` and a public key `id_rsa.pub`. We just need to put the public key into GitHub so they function as deploy keys.
- Go to your forked [Choose Your Own Adventure Presentations](#) repository page on GitHub.
- Click the settings link on the right navigation bar within the forked repository's page.



- Click "Deploy Keys" on the left navigation bar.



- Press the "Add deploy key" button.

13. Two text boxes will appear on the screen. Fill in a title and the contents of the `id_rsa.pub` key file you just created. Make sure you use the `id_rsa.pub` file, not the private key stored in `id_rsa`. Do not allow write access since we won't need that for Jenkins right now.

14. Press the "Add Key" button. Now you're all set to clone this forked repository's code onto your production server.

15. Back on the server as the Jenkins user, let's test out our deploy key. Execute the following command and specify 'yes' if it asks you to verify the host signature.

```
git ls-remote -h ssh://git@github.com/makaimc/choose-your-own-adventure-presentations.git HEAD
```

16. Next we need to load our Ansible playbook onto the Jenkins CI server so the jenkins user can access them. There are a couple of ways to do this.

17. The first option is to use the `securecopy` command from your local machine to put the automation files onto the server. For example, you can `tar` up your local Ansible playbook then `scp` it to the server with the following commands. Run these commands from your local machine and make sure to replace the IP address with the IP address of your Jenkins machine.

```
tar -cvf deploy.tar ~/fsp-deployment-guide  
scp -i
```

18. Then copy those files from the `deployer` user's home directory into the `~` (home directory) of the jenkins user and use the following command to untar the files.

```
tar -xvf deploy.tar.gz
```

19. The other option is to clone the Ansible playbook automated deployment repository. For example, if you want to use the default Ansible playbook you'd clone the following repository. **Important note:** you will have to enter your own values into the configuration and ensure you have the appropriate SSH keys to get this repository to automate your deployments.

```
git clone git@github.com:makaimc/fsp-deployment-guide
```

20. When you get the Ansible playbook on the continuous integration server, continue on with these steps to test the playbook out. Create a virtualenv for the Jenkins account so it can use Fabric and Ansible to perform the deployment.

```
virtualenv cyoa
source cyoa/bin/activate
pip install Fabric==1.10.2 Ansible==1.9.2
```

21. Test out the Ansible playbook while logged into the `jenkins` user. For testing purposes, export the `REMOTE_DEPLOYER_PASSWORD` variable, which is the sudo password for the deployer user on the production server, so that the updated Ansible playbook can pick it up.

```
cd ~/fsp-deployment-guide
export REMOTE_DEPLOYER_PASSWORD='remote deployer password'
./deploy_prod
```

22. You'll have to enter the sudo password of the remote `deployer` user for the server you're performing the deployment on.

23. You'll likely see a message like the following, only with your production server's IP address instead of the one shown in this message. Enter 'yes' and press enter so the deployment can proceed.

```
The authenticity of host '66.175.209.129 (66.175.209.129)' can't be established.
ECDSA key fingerprint is 7a:2a:d0:4c:62:5e:19:e9:ee:48:7b:f2:70:f9:05:c7.
```

```
Are you sure you want to continue connecting (yes/no)? yes
ok: [66.175.209.129]
```

## Creating Our Jenkins Build Job

We need to configure Jenkins through the user interface so that it pulls the project from GitHub and kicks off the Ansible deployment.

1. Click "Create New Job" on Jenkins' main dashboard page.



2. Name it "CYOA" and select "Freestyle Project", then click the "OK" button.

Item name CYOA

**Freestyle project**  
 This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Maven project**  
 Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

**External Job**  
 This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See the documentation for more details.

**Multi-configuration project**  
 Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**OK**

- Under the "Source Code Management" section, select "Git" project and add your credentials that were configured on the command line.

**Source Code Management**

None  
 CVS  
 CVS Projectset  
 **Git** #1: Select Git.  
 Repositories

Repository URL: git@github.com:makaimc/choose-your-own-adventure-presentations.git #2: Enter your fork's repository.

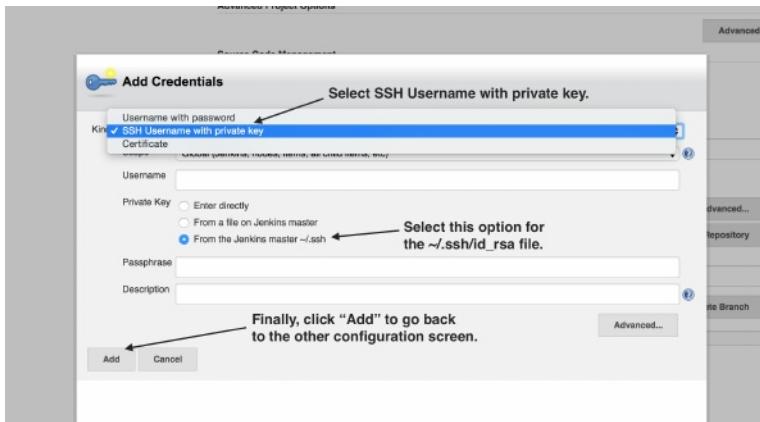
Credentials: jenkins #3 Click Add to go to the "Add Credentials" page.

#4 After adding your credentials, select "jenkins".

Advanced... Add Repository Delete Repository

Branches to build Branch Specifier (blank for 'any') \*/master Add Branch Delete Branch

Repository browser (Auto)



4. Select "Execute shell" and enter the following command on a single line:

```
export REMOTE_DEPLOYER_PASSWORD='remote deployer password'; /fsp-deployment-guide/
prod/deploy.yml --private-key=/fsp-deployment-guide/prod/hosts
```

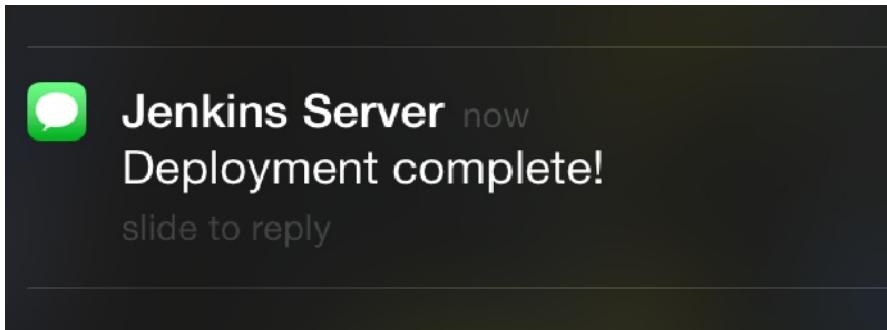


5. Click Save and the project settings will immediately take effect.
6. We've set up the initial settings for the project including pulling the code from the remote GitHub repository onto our server with a new deploy key.

Let's test it manually now. Click the build button on the dashboard to kick off a new build.

The screenshot shows the Jenkins dashboard with a single job listed: 'CYOA'. The job has a blue icon, a yellow sun icon, and the name 'CYOA'. It shows 'Last Success' at '25 min - 81', 'Last Failure' as 'N/A', and 'Last Duration' as '0.71 sec'. Below the table, there is a legend with three RSS feed icons: 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'. To the right of the table, there is a link: 'Click here to manually kick off a build.' with a small 'add description' link next to it. A callout arrow points from the text 'Click here to manually kick off a build.' to the link itself.

If the build is successful, you'll receive a text message a bit later that tells you the deployment is complete!



## Go Ahead, Push Some New Code

Modify some code in your local project, commit the code to the local repository and push it to the remote repository on GitHub. GitHub will then send a service hook HTTP POST request to your Jenkins server that will kick off a build.

Woohoo! Full service automated deployment by the Jenkins continuous integration server is complete!

## Next Steps

Note that Jenkins is only running over HTTP right now - your interactions are not encrypted. There's more security work that needs to be done to lock Jenkins down. The next step to take is to [secure Jenkins behind a reverse proxy that uses HTTPS](#) as we do with our WSGI server that runs our Python web application.

If a full Jenkins security tutorial is something you'd like to see as part of this guide, send me an email at matthew.makai@gmail.com, create [a pull request on the fsp-deployment-guide repository](#) on GitHub. There are many directions this book can continue to expand into and your feedback is valuable for making sure the right content for the community is written.

For more future directions, continue reading into the next chapter, "What's Next?".

## Other Open Source CI Projects

If Jenkins CI server isn't your jam, here is a list of other continuous integration servers that you could also try for your deployment needs.

- [Go CD](#) is a CI server by [ThoughtWorks](#) that was designed with build pipelines and test & release cycles in mind. The [Go CD source code is available on GitHub](#).
- [Strider](#) is a CI server written in node.js. [Strider's source code is also on GitHub](#).

- [BuildBot](#) is a continuous integration **framework** with a set of components for creating your own CI server. It's written in Python and intended for development teams that want more control over their build and deployment pipeline. [The BuildBot source code is on GitHub as well.](#)

## Hosted CI Services

If you want to skip setting up your own continuous integration server, take a look at the following hosted CI services. They make setting up CI on one or more projects easier than deploying your own Jenkins instance, but for some projects the monthly fees may be prohibitive for your budget.

- [Travis CI](#) provides free CI for open source projects and has a [commercial version](#) for private repositories.
- [Bamboo](#) is [Atlassian](#)'s hosted continuous integration that is also free for open source projects.
- [Circle CI](#) works with open or closed source projects on GitHub and can deploy them to Heroku if builds are successful.
- [Shippable](#) uses Docker containers to speed the build and integration process. It's free for public repositories.
- [Drone](#) is another CI service that also provides free builds for open source projects.
- [Codeship](#) provides continuous integration for Python 2.7.
- [Snap](#) is a CI server and build pipeline tool for both integrating and deploying code.

## Additional Continuous Integration Resources

The CI set up we created in this chapter is just the beginning for deployment automation. These articles should give you further insight into CI and where you could take the infrastructure from here.

- [What is continuous integration?](#) is a classic article by Martin Fowler on the concept of CI and how to implement it.
- [Diving into continuous integration as a newbie](#) is a retrospective on learning CI from a Rackspace intern on how she learned about CI.
- [Continuous Deployment For Practical People](#) provides a broad overview of the subject.
- [Continuous Integration & Delivery - Illustrated](#) uses well done drawings to show how continuous integration and delivery works for testing and managing data.
- [Thoughts on web application deployment](#) walks through stages of deployment with source control, planning, continuous deployment and monitoring the results.
- [Practical continuous deployment](#) defines delivery versus deployment and walks through a continuous deployment workflow.

**Chapter 11**

# **What's Next?**

# What's Next?

Our deployment is now automated and runs continuously whenever code is committed and pushed into the remote Git repository. However, there are many enhancements we could make to this deployment process.

Most large-scale web application deployments use a suite of further techniques that increase the capabilities of their infrastructure. This chapter is intended as a “future directions” you may want to take in your own deployment. I’ll also add additional chapters to this book on any of these topics based on reader feedback.

## Deployment Enhancements

Our deployment process right now is simple - it’ll get your application out of source control and onto a server that can execute the Python code. Yet there are a whole bunch of ways to make it better by incorporating additional steps into the Ansible playbook. Here’s a list of potential concepts to research and consider adding to our deployment:

- Add test servers before deploying to production
- Allow source code roll backs after a bad deployment
- Incorporate unit and integration tests into the build process
- Rotate the public-private key pairs for additional security
- Back up the server on a regular schedule

- Deploy through a system package instead of version control

## Improving Performance

There are numerous ways the web application's performance can be improved by modifying the server configuration. Here are several possibilities for scaling up server and application performance:

- Resize up the Linode virtual private server
- Split the application and database servers
- Incorporate load balancers with multiple web servers
- Serve static assets from a content delivery network (CDN)
- Replicate the database to a read-only copy

## Onward!

Those are a few options for improving the deployment process and upgrading the application's performance. What content would you like to see added to the book? Send me an email at matthew.makai@gmail.com or tweet me [@mattmakai](#) or [@fullstackpython](#) and let me know what you think. All updates to this book will be available for free to current purchasers, so your opinion is meaningful in determining how the content grows over time!

**Appendix A**

# Glossary

# Glossary

This glossary provides plain language definitions of the concepts found throughout this book. If you're uncertain about what a word or concept means, this glossary plus the additional information provided on [Full Stack Python](#) should clear up your confusion. If it doesn't, email me and I'll make sure to clarify the concepts in future version of the book.

## Application dependencies

*Application dependencies* are the libraries other than your own project code that are required to create and run your application. These dependencies are referenced in a `requirements.txt` file, retrieved from [PyPI](#) via the `pip` command and installed into a `virtualenv` to isolate the dependencies from other project's dependencies.

## Continuous integration

*Continuous integration* is the concept of building and testing code upon every change a developer checks into version control. The theory is that if code is not integrated together until the end of the project, the integration period will be painful due to numerous bugs that are created at the edges of where the software components are pieced together. Continuous integration server implementations include [Jenkins](#), [TeamCity](#) and [Go](#).

# Deployment

A *deployment* is the process of packaging up an application's code and putting it on a server where it can be run for users. Some deployments are just to a development server for fellow developers to test their code, while other deployments go to a test server to make sure the code works properly. The last stage in a deployment is usually to one or more production servers that host the canonical version of an application.

# Flask

[Flask](#) is a Python web framework created with a small core and easy-to-extend philosophy. Flask pieces together several open source projects, including [Werkzeug](#), [Jinja2](#) and [SQLAlchemy](#) (for relational database object-relational mapping) to make it easier to build WSGI-compatible web applications with the framework. Flask is the web framework used by the [Choose Your Own Adventure Presentations](#) example application deployed throughout this book.

# NoSQL

*NoSQL*, also referred to as *Not Only SQL*, is a database type that forgoes one or more constraints present in traditional relational databases in order to gain some other attribute - often speed of read or write access. There are many types of [NoSQL](#) [database](#) implementations, including as [Redis](#), [MongoDB](#), [Cassandra](#) and [HBase](#).

## **pip**

*Pip* is a tool used to install Python packages from one location, often a remote dependency management server such as [PyPI](#) or [GitHub](#), into a local Python installation directory. Pip is often used with `requirements.txt` file so the appropriate versions of application dependencies for an application can be installed.

## **PostgreSQL**

[PostgreSQL](#) is an open source [relational database](#) implementation. In this book, PostgreSQL is the primary data storage backend for the application.

## **Production server**

A *production server* or *production environment* hosts the canonical version of the running code for your application. There can be many environments for development and testing, but production is the only one that contains the live version of your application.

## **PyPI**

The *Python Package Index* also known as *PyPI*, is a central remote server that contains numerous versions of application dependencies for Python packages that were uploaded by their authors. For example, the Flask web microframework [is stored on PyPI](#) and can be retrieved to a local computer by using the `pip` command. PyPI is

pronounced “Pie-pee-eye” to differentiate it from the [PyPy](#) project, which is a separate concept not covered in this book.

## Source control

*Source control*, also known as *version control*, stores software code files with a detailed history of every modification made to those files. Source control systems allow developers to modify code without worrying about permanently screwing something up. Unwanted changes can be easily rolled back to previous working versions of the code, which makes developing in both individual and team environments easier. Examples of source control implementations include [Git](#), [Mercurial](#) and [Subversion](#).

## Task queue

A *task queue* is a concept for a piece of software that executes work in the background outside the HTTP request-response cycle of a web application. The work performed by a task queue would often take too long to process when a HTTP request came in or would need to be performed on a regular interval and are not initiated by HTTP requests. Python task queue implementations include [Celery](#), [RQ](#) and [Taskmaster](#).

## **virtualenv**

*Virtualenv* provides application dependency isolation for Python projects. The virtualenv creates a separate copy of the Python installation that is clean of existing code libraries and provides a directory for new application dependencies on a per-project basis (a programmer can technically use a virtualenv for many projects at once but that practice is discouraged).

## **Web framework**

*Web frameworks* are code libraries that make it easier to build web applications. Web frameworks take care of common web development issues, such as HTTP requests and responses, URL routing, authentication, data handling and manipulation, sessions and security. The web framework implementation in this book is [Flask](#). Other common Python-based web frameworks include [Django](#), [Bottle](#), [Pyramid](#) and [Morepath](#).

## **Web server**

A *web server* is an application that receives and responds to [Hypertext Transport Protocol](#) (HTTP) requests. The web server implementation used in this book is [Nginx](#). Nginx serves static and provides a reverse proxy, which is an intermediate handler between a web client and the WSGI server running our web application. Other examples of web server implementations include the [Apache HTTP Server](#), [IIS](#).

# WSGI

*Web Server Gateway Interface (WSGI)* is a standard interface between Python web applications or frameworks and web servers. The interface was created to promote web application portability between different WSGI server implementations such as [Green Unicorn](#), [uWSGI](#), [wsgiref](#) and [mod\\_wsgi](#).

**Appendix B**

# **More Python Resources**

# More Python Resources

This appendix wraps up the book with a collection of the best Python resources I've found. Many of these resources are free and the ones that cost money are well worth the investment to purchase them.

## The Best Python Resources

The Python community is amazing at sharing detailed resources and helping beginners learn to program with the language. There's so many resources out there though that it can be difficult to know how to find them.

This chapter aggregates the best Python resources with a brief description of its one's learning purpose.

## New to Programming

If you're learning your first programming language these books were written with you in mind. Developers learning Python as a second or later language should skip down to the next section for "experienced developers".

- To get an introduction to both Python and Django at the same time, consider purchasing [Real Python](#) by Fletcher, Michael, and Jeremy.

- [Learn Python the Hard Way](#) is a free book by Zed Shaw for learning Python programming.
- [Dive into Python 3](#) is an open source book provided under the Creative Commons license and available in HTML or PDF form.
- [Python for You and Me](#) is an online book for people unfamiliar with the Python programming language.
- [A Byte of Python](#) is a beginner's tutorial for Python.
- Code Academy has a [Python track](#) for people completely new to programming.
- There's a Udacity course by one of the creators of Reddit that shows how to [use Python to build a blog](#). It's a great introduction to web development concepts through coding.
- The O'Reilly book [Think Python: How to Think Like a Computer Scientist](#) available in HTML form for free on the web.
- [Python Books](#) is a collection of freely available books on Python, Django, and data analysis.

## Experienced Developers New to Python

People with prior software development experience will have an easier time learning the Python language than those who've never programmed before. In those cases it makes sense to use resources that spend less words on computer science fundamentals and more effort on explaining the specifics of the Python syntax. The following resources do a good job of teaching Python to those who already program in other other languages.

- [Learn Python in Y minutes](#) provides a whirlwind tour of the Python language. The guide is especially useful if you're coming in with previous software development experience and want to quickly grasp how the language is structured.
- [The Hitchhiker's Guide to Python](#) contains a wealth of community-curated information both on the Python language and the ecosystem around it.
- [Google's Python Class](#) contains lecture videos and exercises for learning Python.

## Beyond the Basics

Whether you're a newcomer to programming or a grizzled veteran, there's always room for you to grow and more to learn after grasping the Python syntax. The following resources are useful for understanding the broader ecosystem and more

advanced subjects.

- [The Python Ecosystem: An Introduction](#) provides context for virtual machines, Python packaging, pip, virutalenv and many other topics after learning the basic Python syntax.
- The [Python Reddit](#) and [Learn Python Reddit](#) roll up great Python links and have an active community ready to answer questions from beginners and advanced Python developers alike.
- [Good to Great Python Reads](#) is a collection of intermediate and advanced Python articles around the web focused on nuances and details of the language itself.

- The blog [Free Python Tips](#) provides posts on Python topics as well as news for the Python ecosystem.

## Videos, Screencasts and Presentations

If you prefer to watch videos and screencasts instead of reading through more text, these are some great resources for you to learn more about Python.

- [Kate Heddleston](#) gave a talk at PyCon 2014 called [Full-stack Python Web Applications](#) with clear visuals for how numerous layers of the Python web stack fit together. There are also [slides available from the talk](#) with all the diagrams.
- My [EuroPython 2014 "Full Stack Python"](#) talk goes over each topic from this guide and provides context for how the pieces fit together. The [talk slides](#) are also available.
- [Neckbeard Republic](#) provides free screencasts for learning intermediate topics. I typically prefer to learn by reading. However, these videos are helpful in seeing the code on the screen instead of just looking at static code snippets.
- [PyVideo](#) organizes and indexes thousands of Python videos from both major conferences and meetups.

## Curated Python Packages Lists

The following Git repositories aggregate open source libraries into categories, such as Natural Language Processing and web application frameworks.

- [awesome-python](#) is a huge list of Python frameworks, libraries and software.
- [easy-python](#) is like awesome-python although instead of just a Git repository this site is in the Read the Docs format.
- [awesome-django](#) provides only libraries related to the Django web application framework.

## Newsletters

Email newsletters are good for keeping up with the latest open source projects and articles. Rather than you having to spend your time searching the web for these

helpful resources, you can have someone else aggregate them for you and deliver to your inbox.

- [Python Weekly](#) provides a free weekly roundup of the latest Python articles, videos, projects and upcoming events.
- [PyCoder's Weekly](#) is another great free weekly email newsletter similar to Python Weekly. The best resources are generally covered in both newsletters but they often cover different articles and projects from around the web.
- [Import Python](#) is a newer newsletter than Python Weekly and PyCoder's Weekly. So far I've found this newsletter covers different sources from other newsletters. It's well worth subscribing to all three so you don't miss

anything.

- The [Full Stack Python monthly newsletter](#) is a monthly newsletter that focuses on a single topic each month. For example, one month will aggregate great Flask resources, while another month will provide WSGI server configurations.

**Appendix C**

# **App Code Tutorial**

# App Code Tutorial

This chapter is a streamlined version of the original code tutorial for the example application we deployed in this book. This [Choose Your Own Adventure Presentations Flask web app](#) provides a reasonable example for deployment because it is open source and relies on many widely used web application technologies such as PostgreSQL, Redis, Celery, WebSockets and Green Unicorn.

Going through this appendix is not required to do the deployment in this book. It's included since I wrote the original posts for the [Twilio blog](#).

Note that this tutorial will take you up to the [tutorial-step-6](#) tag but I kept working on the project so for this book you should use the [master branch](#) for the deployment.

# Choose Your Own Adventure Presentations with Reveal.js, Python and WebSockets



You're preparing a technical talk on your new favorite open source project to present to your local software meetup group.

---

*How do you proceed? If you choose to create another passe linear slide deck, load up Microsoft PowerPoint. If you decide to build a childhood nostalgia-packed Choose Your Own Adventure presentation, continue reading this tutorial.*

---

Good choice! To create our Choose Your Own Adventure presentation we'll use [Reveal.js](#), [Flask](#), [WebSockets](#) and [Twilio SMS](#). Our final product will have decision screens like the following screenshot where SMS votes from the audience are counted and displayed in real-time to determine the next step in the presentation story. Once

the votes are tallied the presenter can click the left choice or the right choice to go to the appropriate next slide in the presentation.



Take a look at the [DjangoCon US 2014 "Choose Your Own Django Deployment Adventure"](#) video if you want to see an example of a Choose Your Own Adventure presentation in action.

## What we'll need

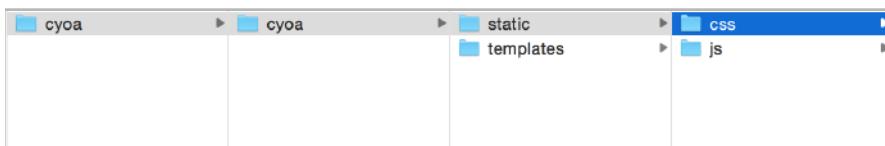
- [Reveal.js](#) for browser-based presentations
- [Python](#) with application dependencies:
  - [Flask](#)
  - [Flask-SocketIO](#) for handling websockets with Flask

- [Flask-Script](#) for utility commands such as running our Flask server
- [Gunicorn](#) to run our server
- [redis-py](#) for interacting with Redis
- [Redis](#) for calculating results and short-term vote storage
- A [Twilio account](#) with an SMS number so audiences can vote which path the presentation should follow
- [Ngrok](#) for a secure tunnel to our local server running the presentation

Do you want to skip typing out the code yourself? Check out [this open source repository on GitHub](#). The Git repository contains the final code as well as intermediate tags named [tutorial-step-1](#), [tutorial-step-2](#) and [tutorial-step-3](#) for each section below.

## Building the Presentation

Let's first create the directory structure necessary for our project. The nested subdirectories will look like the following screenshot. Our base directory name and Flask app directory name will be **cyoa**, an acronym for "Choose Your Own Adventure."



At the base directory of our project, `cyoa/`, create a file named `requirements.txt` with the following content in it. Each line contains a dependency for our project that we'll install in a moment.

`cyoa/requirements.txt`

```
Flask==0.10.1
Flask-Script==2.0.5
Flask-SocketIO==0.4.1
gunicorn==19.1.1
redis==2.10.3
twilio==3.6.8
```

This application uses Flask to serve up the presentation. The [Flask-Script extension](#) will assist us in creating a `manage.py` file with helper commands to run the web application.

The [Flask-SocketIO extension](#) handles the server-side websockets communication.

Create a new virtualenv outside the base project directory to separate your Python dependencies from other apps you're working on.

```
virtualenv cyoa
```

Activate the virtualenv before we install the dependencies.

```
source cyoa/bin/activate
```

To install these dependencies run the following `pip` command on the command line.

```
pip install -r requirements.txt
```

Create a file named `cyoa/manage.py` with the following contents.

```
cyoa/manage.py

from gevent import monkey
monkey.patch_all()

import os

from cyoa import app, redis_db, socketio
from flask.ext.script import Manager, Shell

manager = Manager(app)

def make_shell_context():
    return dict(app=app, redis_db=redis_db)

manager.add_command("shell", Shell(make_context=make_shell_context))
```

```

@manager.command
def runserver():
    socketio.run(app, "0.0.0.0", port=5001)

if __name__ == '__main__':
    manager.run()

```

The above `manage.py` file will help us run our Flask application with the `python manage.py runserver` command once we've put more of the pieces in place.

Next create a file `cyoa/cyoa/config.py` (the `cyoa/cyoa` subdirectory is where most of our Python code will live other than `manage.py`) and add the below code to create the configuration variables we'll need for our Flask application.

#### `cyoa/cyoa/config.py`

```

import os

# General Flask app settings
DEBUG = os.environ.get('DEBUG', None)
SECRET_KEY = os.environ.get('SECRET_KEY', None)

# Redis connection
REDIS_SERVER = os.environ.get('REDIS_SERVER', None)
REDIS_PORT = os.environ.get('REDIS_PORT', None)
REDIS_DB = os.environ.get('REDIS_DB', None)

# Twilio API credentials
TWILIO_ACCOUNT_SID = os.environ.get('TWILIO_ACCOUNT_SID', None)
TWILIO_AUTH_TOKEN = os.environ.get('TWILIO_AUTH_TOKEN', None)
TWILIO_NUMBER = os.environ.get('TWILIO_NUMBER', None)

```

Here's a rundown of what each of these environment variables specified in `config.py` is for:

- **DEBUG** - True or False for whether Flask should display error messages if something goes wrong
- **SECRET\_KEY** - a long key that should be kept secret

- **REDIS\_SERVER**- in this case likely to be localhost or wherever Redis is running
- **REDIS\_PORT**- generally set to 6379 for the default Redis port
- **REDIS\_DB**- set to 0
- **TWILIO\_ACCOUNT\_SID** found on your [Twilio account dashboard](#)
- **TWILIO\_AUTH\_TOKEN** also found on your [Twilio account dashboard](#)
- **TWILIO\_NUMBER** a number you've [purchased on Twilio](#)

Set the **DEBUG** and **SECRET\_KEY** environment variables now. The Redis and Twilio environment variables will be set in the next section.

Setting up environment variables depends on your operating system. There are guides for every major operating system, whether that is [Ubuntu Linux](#), [Mac OS X](#) or [Windows](#).

The next file we need to create is **cyoa/cyoa/\_\_init\_\_.py** which will set up the core pieces for our the Flask app.

#### **cyoa/cyoa/init.py**

```
from flask import Flask
from flask.ext.socketio import SocketIO
import redis

app = Flask(__name__, static_url_path='/static')
app.config.from_pyfile('config.py')

from config import REDIS_SERVER, REDIS_PORT, REDIS_DB

redis_db = redis.StrictRedis(host=REDIS_SERVER, port=REDIS_PORT, db=REDIS_DB)

socketio = SocketIO(app)

from . import views
```

Next create a file named **cyoa/cyoa/views.py** and put the following code inside.

### cyoa/cyoa/views.py

```
from flask import render_template, abort
from jinja2 import TemplateNotFound

from . import app

@app.route('/<presentation_name>', methods=['GET'])
def landing(presentation_name):

    try:
        return render_template(presentation_name + '.html')
    except TemplateNotFound:
        abort(404)
```

For now the above file contains a single view named `landing`. This view `landing` obtains a presentation name from the URL and checks the `cyoa/cyoa/templates` directory for an HTML template file matching that name. If a matching file is found, `landing` renders the HTML template. If no file name matches the presentation name in the URL, `landing` returns a 404 HTTP response.

We're awfully close to getting to fire up this presentation to take a look at it. An HTML template file along with some static files are necessary for displaying the presentation. Create a Reveal.js presentation in templates directory named `cyoa/cyoa/templates/cyoa.html`. Add the following HTML inside this file.

### cyoa/cyoa/templates/cyoa.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Choose Your Own Adventure Presentations with Reveal.js!</title>
        <meta name="apple-mobile-web-app-capable" content="yes" />
        <meta name="apple-mobile-web-app-status-bar-style" content="black-translucent" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
user-scalable=no, minimal-ui">
        <link rel="stylesheet" href="/static/css/reveal.css">
        <link rel="stylesheet" href="/static/css/default.css" id="theme">
        <link rel="stylesheet" href="/static/css/zenburn.css">

        <!--[if lt IE 9]>
```

```

<script src="lib/js/html5shiv.js"></script>
<![endif]-->
</head>
<body>
  <div class="reveal">
    <div class="slides">
      <section>
        <h1>Choose Your Own Adventure!</h1>
        <h3>With SMS voting</h3>
      </section>

      <section>
        <div>
          <h2>Text "left" or "right" to</h2>
          <!-- replace this with your Twilio number -->
          <h1>(xxx) 555-1234</h1>
        </div>
        <br>
        <div style="display: inline;">
          <div style="float: left;">
            <h2><a href="#/2">left</a></h2>
            <h1><div id="left">0</div> votes</h1>
          </div>
          <div style="float: right;">
            <h2><a href="#/3">right</a></h2>
            <h1><div id="right">0</div> votes</h1>
          </div>
        </div>
      </section>

      <!-- linked from first choice -->
      <section>
        <h1>Left</h1>
      </section>

      <!-- linked from second choice -->
      <section>
        <h1>Right</h1>
      </section>
    </div>
  </div>

```

```

<script type="text/javascript"
       src="http://code.jquery.com/jquery-1.11.1.min.js"></script>
<script type="text/javascript"
       src="//cdnjs.cloudflare.com/ajax/libs/socket.io/0.9.16/socket.io.min.js">
</script>
<script src="/static/js/reveal.js"></script>

<script>
  Reveal.initialize({
    controls: true,
    progress: true,
    history: true,
    center: true,
    theme: Reveal.getQueryHash().theme,
    transition: Reveal.getQueryHash().transition || 'default',
    dependencies: []
  });
</script>
</body>
</html>

```

As you can see in the HTML there are three locally hosted static CSS files and a JavaScript file you'll need for your project. [Download these files from GitHub](#) or

download and extract this `cyoa.tar.gz` archive into the `cyoa/cyoa/static` directory. The CSS and JavaScript files we need are the following:

- `js/reveal.js`- code for creating the beautiful Reveal.js presentation
- `css/default.css`- default style for Reveal.js decks
- `css/reveal.css`- required for running Reveal.js presentations properly
- `css/zenburn.css` necessary for syntax highlighting within a presentation

With these files in place we can try out the presentation to make sure everything so far is in working order. Fire up Flask with the following command.

```
python manage.py runserver
```

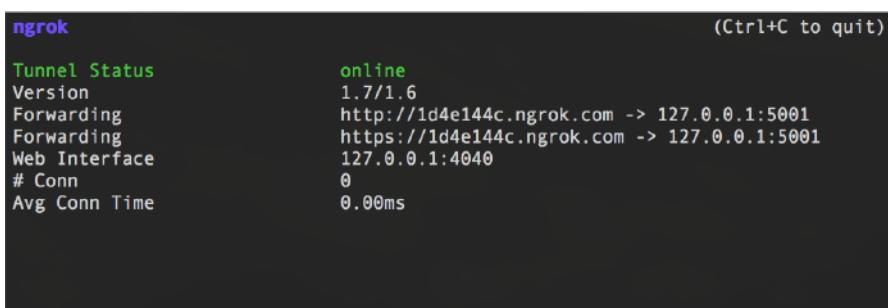
Now we can view the initial version of the presentation by going to <http://localhost:5001/cyoa/> in the browser. Flask will find the `cyoa.html` file in the `templates` directory and serve that up so you can view it.

However, the presentation will only come up on your own computer. To access the presentation from other computers you'll need to deploy to a hosting server or create a localhost tunnel using a tool such as [Ngrok](#). Sign up for Ngrok and [download the Ngrok tunneling application](#).

Fire up ngrok on port 5001 where our Flask application is running. See this [configuring Ngrok post](#) if you're running Windows. On Linux and Mac OS X Ngrok can be run with the following command when you're in the directory that ngrok is located in.

```
./ngrok 5001
```

You'll see a screen like the following. Take note of the unique https:// forwarding URL as we'll need that again in a minute for the Twilio webhook.



```
ngrok
(TCtrl+C to quit)

Tunnel Status: online
Version: 1.7/1.6
Forwarding: http://1d4e144c.ngrok.com -> 127.0.0.1:5001
Forwarding: https://1d4e144c.ngrok.com -> 127.0.0.1:5001
Web Interface: 127.0.0.1:4040
# Conn: 0
Avg Conn Time: 0.00ms
```

You can now pull the presentation both from the localhost URL as well as the forwarding URL set up by ngrok.

## Accepting SMS votes

So far we've written the Python code to serve a Reveal.js presentation and exposed it via a localhost tunnel. Now we need a way for the audience to vote on which path they want the presentation to go. To do that we'll show a Twilio phone number on the screen and use Twilio SMS to let the audience vote. If you've already got an account grab a new or existing phone number, otherwise [sign up and upgrade a Twilio account](#).

We'll need to set up the message webhook so that each vote SMS to the Twilio number is sent to our Flask app. The Twilio [number screen](#) should look like the following.

The screenshot shows the Twilio Number properties page for a number (202) 759-0512. A green annotation highlights the 'Webhook sends HTTP POST to URL when inbound SMS votes come in' section. A green arrow points from this annotation to the 'Request URL' field under the 'Messaging' tab, which contains the URL <https://1d4e144c.ngrok.com/cyoa/twilio/webhook/>. Other visible fields include 'Voice' settings with a Request URL of <https://demo.twilio.com/welcome/voice/>, and a 'Save' button at the bottom.

Paste in your Ngrok forwarding URL, which will look like "https://unique Ngrok code.ngrok.com", along with "/cyoa/twilio/webhook/" to the Messaging Request URL

text box then hit save. For example, if your Ngrok tunnel is "1d4e144c" your URL to paste into the webhook text box should look like the following URL.

<https://1d4e144c.ngrok.com/cyoa/twilio/webhook>

Now Twilio will send a POST HTTP request to the ngrok port forwarding URL which will be sent down to your localhost server.

Make sure to update the phone number in [you](#)**cyoa/cyoa/templates/cyoa.html** presentation file. Look for the line like the following and replace it with your Twilio number.

**cyoa/cyoa/templates/cyoa.html**

```
<!-- update your number here -->
(xxx) 555-1234
```

However, we're not yet ready to accept those incoming SMS votes. First we need to ensure Redis is running and write Python code to persist votes to it.

Ensure that the Redis server is installed on your system since that will be our temporary vote storage system. There's a[quickstart installation guide available](#) for your operating system of choice if you're unfamiliar with Redis. In that quickstart you just need to go from the beginning through the "Starting Redis" section and you'll be set for finishing this project.

Let's add code in our**cyoa/cyoa/views.py**file for the Twilio webhook HTTP POST that occurs on an inbound SMS. We'll increment a counter for each keyword texted in by people in the audience.

**cyoa/cyoa/views.py**

```
import cgi
from flask import render_template, abort, request
from jinja2 import TemplateNotFound
from twilio import twiml
```

```

from twilio.rest import TwilioRestClient
from .config import TWILIO_NUMBER
from . import app, redis_db

client = TwilioRestClient()

@app.route('/<presentation_name>', methods=['GET'])
def landing(presentation_name):
    try:
        return render_template(presentation_name + '.html')
    except TemplateNotFound:
        abort(404)

@app.route('/cyoa/twilio/webhook', methods='POST')
def twilio_callback():
    to = request.form.get('To', '')
    from_ = request.form.get('From', '')
    message = request.form.get('Body', '').lower()
    if to == TWILIO_NUMBER:
        redis_db.incr(cgi.escape(message))
        resp = twiml.Response()
        resp.message("Thanks for your vote!")
    return str(resp)

```

It'd be useful to have a way to clear out votes from Redis in between rehearsals for our presentation. Update `cyoa/manage.py` with a new function to clear votes from Redis.

#### `cyoa/manage.py`

```

from gevent import monkey
monkey.patch_all()

```

```

import os

import redis

from cyoa import app, redis_db, socketio
from flask.ext.script import Manager, Shell

manager = Manager(app)

def make_shell_context():
    return dict(app=app, redis_db=redis_db)

manager.add_command("shell", Shell(make_context=make_shell_context))

@manager.command
def runserver():
    socketio.run(app, "0.0.0.0", port=5001)

@manager.command
def clearedis():

    rediscli = redis.StrictRedis(host='localhost', port=6379, db=0)

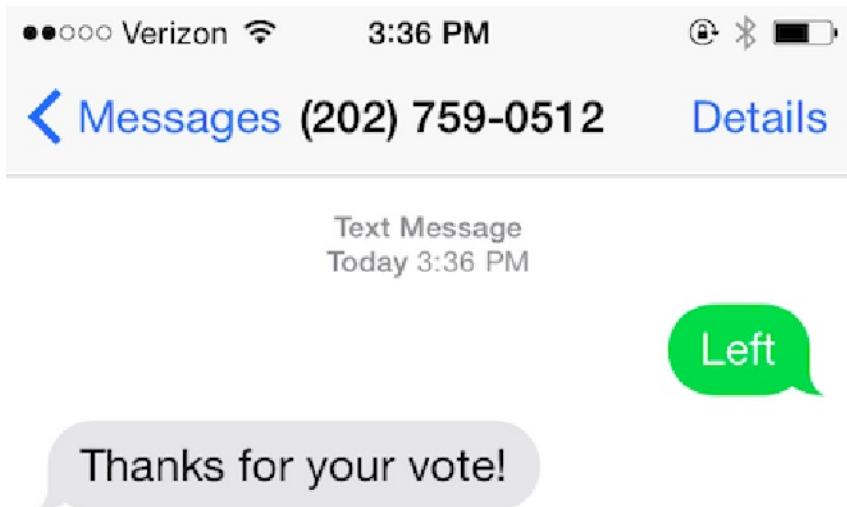
    rediscli.delete('left')
    rediscli.delete('right')

if __name__ == '__main__':
    manager.run()

```

If Flask and ngrok are still running we can test this out now, otherwise fire them up and let's text either "left" or "right" to your Twilio number.

When you test out SMS inbound text messages you should receive the "Thanks for your vote!" response like in the screenshot below.



At this point those inbound text message votes are also being stored in Redis. We're so close to wrapping up this code at this point let's write a bit more Python to test that the messages are indeed in Redis and propagate the results to the presentation via websockets.

## Adding WebSockets for Real-time Updates

We need one more new Python file to handle the server-side websockets communication. Websockets allow streams of communication between the client and server. In our presentation that allows the vote counts to be updated on the presentation slide as soon as the server emits a new message. Add a file named `cyoa/cyoa/websockets.py` with the following code to set up the server-side websockets implementation.

```

websockets.py

from flask.ext.socketio import emit

from . import socketio

@socketio.on('connect', namespace='/cyoa')
def test_connect():
    pass

@socketio.on('disconnect', namespace='/cyoa')
def test_disconnect():
    pass

```

The above code in the `update_count` function allows the websocket clients, in this case the browser that loads the presentation, to connect to the websockets stream and receive future messages.

Add one line to the bottom of `cyoa/cyoa/__init__.py` to import the `cyoa/cyoa/websockets.py` file we just wrote.

`cyoa/cyoa/init.py`

```

from flask import Flask
from flask.ext.socketio import SocketIO
import redis

app = Flask(__name__, static_url_path='/static')
app.config.from_pyfile('config.py')

from config import REDIS_SERVER, REDIS_PORT, REDIS_DB

redis_db = redis.StrictRedis(host=REDIS_SERVER, port=REDIS_PORT, db=REDIS_DB)

socketio = SocketIO(app)

from . import views

from . import websockets

```

Update `cyoa/cyoa/views.py` with the following highlighted lines. We're adding the `socketio.emit` call so votes are passed via websockets to the presentation.

#### `cyoa/cyoa/views.py`

```
import cgi
from flask import render_template, request, abort
from jinja2 import TemplateNotFound
from twilio import twiml
from twilio.rest import TwilioRestClient

from .config import TWILIO_NUMBER
from . import app, redis_db

from . import socketio

client = TwilioRestClient()

@app.route('/<presentation_name>', methods=['GET'])
def landing(presentation_name):
    try:
        return render_template(presentation_name + '.html')
    except TemplateNotFound:
        abort(404)

@app.route('/cyoa/twilio/webhook', methods=['POST'])
def twilio_callback():
    to = request.form.get('To', '')
    from_ = request.form.get('From', '')
    message = request.form.get('Body', '').lower()
    if to == TWILIO_NUMBER:
        redis_db.incr(cgi.escape(message))

        socketio.emit('msg', {'div': cgi.escape(message),
                             'val': redis_db.get(message)},
                      namespace='/cyoa')

    resp = twiml.Response()
    resp.message("Thanks for your vote!")


```

```
return str(resp)
```

Add the highlighted code below to the `cyoa/cyoa/templates/cyoa.html` presentation file. This code relies on the websockets JavaScript library and JQuery that was included with the file when we srinally created it.

`/cyoa/cyoa/templates/cyoa.html`

```
<script type="text/javascript">

    Reveal.initialize({
        controls: true,
        progress: true,
        history: true,
        center: true,
        theme: Reveal.getQueryHash().theme,
        transition: Reveal.getQueryHash().transition || 'default',
        dependencies: [
        ]
    });

$(document).ready(function() {
    namespace = '/cyoa';

    var socket = io.connect('http://' + document.domain + ':' + location.port + namespace);

    /* add and edit choices here */

    var appropriate_choices = ['left','right',];

    socket.on('msg', function(msg) {
        /* ensure valid vote and div exists */
        if (appropriate_choices.indexOf(msg.div) >= 0) {

            var checkDiv = $('#' + msg.div);

            if (checkDiv.length > 0) {
```

```
    checkDiv.html(msg.val);

  }
}

});

});

</script>
```

Fire up Flask with Ngrok again. Check out <http://localhost:5001/cyoa/> on your browser and you'll see the same presentation as before. However, text your vote of 'left' or 'right' to your number and the votes will appear like in the screenshot below without any page refresh.

## Wrapping it up

You now have a Flask app that can serve up Reveal.js presentations where the audience can interact with the slide on screen by texting in their votes.

Now it's up to you to create the Choose Your Own Adventure content!

# Choose Your Own Adventure Presentations: Wizard Mode Part 1 of 3

You've coded your way through the harrowing challenges of the first [Choose Your Own Adventure Presentations tutorial](#). A sign lies in the road ahead. "PyCon has summoned you to give a [Choose Your Own Adventure talk in Montreal](#)!" it reads.

---

*How do you proceed? If you choose to run from the PyCon challenge, close the browser window now. If you accept the challenge, prepare yourself for the dangers ahead with the new Wizard Mode functionality and continuing reading this blog post.*

---

You're still here, adventurer! Let's get to work. In this series of three blog posts we're going to expand the [Choose Your Own Adventure Presentations application](#) with new a Wizard Mode. If you haven't yet worked through the [srcinal blog post](#) I highly recommend doing that before working through this series. The section below named "A Clean Starting Point" will get you set up for this tutorial if you've already gone through [the srcinal Choose Your Own Adventure Presentations post](#) and just need a fresh copy of the code.

What's this mysterious Wizard Mode we're building? Think of it as an administrative interface that grants you, the presenting wizard, with new magical powers to control your presentations and how the audience can vote for story choices. The interface will only be accessible by authorized wizards through the sign-in page which will look like the following screenshot.



Once you're inside the application you'll be able to manage one or more Choose Your Own Adventure Presentations with a simple screen like this one:

The screenshot shows a web application interface. At the top, there are navigation links: "Wizards Only", "Presentations", and "Sign out". Below this is a section titled "Presentations" with a table listing four entries:

Name	Is Active?	Voting number	Web browser voting?	Story choices
PyCon 2015	True		True	Manage choices
DjangoCon 2014	True	+15039463662	False	Manage choices
Django District March 2015	True	+12027966933	False	Manage choices
Net Promoter Score	True	+12027336150	True	Manage choices

At the bottom left of the table area is a blue button labeled "New Presentation".

As we go about building our new Wizard Mode you'll learn about Flask [form handling](#), [WebSockets](#) and how to persist presentation data to a [PostgreSQL database](#)

There are three posts in this tutorial series where we will incrementally build out functionality:

1. **Wizards Only**: this blog post, where we'll create a section of the application only authorized wizards can access.
2. **Even Wizards Need Web Forms**: the next blog post where we expand the wizard only pages to control our presentations
3. **Voting with a Wand, or Smartphone**: our third and final post where we add a new magical trick to our presentations – voting via web browser when poor cell service prevents SMS voting

At the end of each post we'll be able to test what we just built to make sure it's working properly. If something goes wrong while you're working through the tutorial, the Git tag [tutorial-step-4](#) tag has the end result for code written in this post.

Let's get to work!

## What We'll Need

We'll continue using open source code to build our project including the [existing Choose Your Own Adventure Presentations code base](#) PostgreSQL and several additional Python libraries. Don't worry about downloading anything just yet - we will grab these tools throughout the tutorial. This list is just so you know ahead of time what'll be installed along the way:

- CYOA Presentations repository at the [tutorial-step-3](#) tag stage
- PostgreSQL for persistent storage
- The [psycopg2](#) Python driver to connect to PostgreSQL
- [Flask-login](#) for authentication

- [Flask-WTF](#) for web form handling

Finally, time to roll up the magical cloak sleeves again and dive into coding.

## Wizards Only

We're going to create the wizard's panel to control multiple presentations and how our audience can interact with them. First though we need a way to sort wizards from non-wizards in the form of a sign-in screen.

We'll use four new Python code libraries in this post so let's get those added to our project. These new libraries are highlighted below. Update the requirements.txt file so it matches the following code.

```
Flask==0.10.1
Flask-Script==2.0.5
Flask-SocketIO==0.4.1

Flask-Login==0.2.11
Flask-SQLAlchemy==2.0
Flask-WTF==0.10.3

gunicorn==19.1.1
redis==2.10.3
twilio==3.6.8

psycopg2==2.5.4
```

Make sure to active your project's virtualenv and install the new dependencies with pip using these two steps:

```
source ~/Envs/cyoa/bin/activate
pip install -r requirements.txt
```

With our new dependencies in place we can create our Wizards Only sign-in page.

Create a new file named `models.py` within the `cyoa/cyoa` subdirectory to store information about wizards. The file should have the following contents:

```
cyoa/cyoa/models.py

from flask.ext.login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash

from . import db

class Wizard(UserMixin, db.Model):
    """
        Represents a wizard who can access special parts of the application.
    """

    __tablename__ = 'wizards'
    id = db.Column(db.Integer, primary_key=True)
    wizard_name = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))

    def __init__(self, wizard_name, password):
        self.wizard_name = wizard_name
        self.password = password

    @property
    def password(self):
        raise AttributeError('password is not readable')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)

    def __repr__(self):
        return '' % self.wizard_name
```

Next we need to modify the `cyoa/cyoa/__init__.py` file to create a login manager. The login manager will allow us to control access to wizard settings.

```

cyoa/cyoa/init.py

import redis
from flask import Flask

from flask.ext.login import LoginManager

from flask.ext.sqlalchemy import SQLAlchemy

from flask.ext.socketio import SocketIO

app = Flask(__name__, static_url_path='/static')
app.config.from_pyfile('config.py')

from config import REDIS_SERVER, REDIS_PORT, REDIS_DB

redis_db = redis.StrictRedis(host=REDIS_SERVER, port=REDIS_PORT, db=REDIS_DB)

socketio = SocketIO(app)

db = SQLAlchemy(app)

login_manager = LoginManager()

login_manager.login_view = 'sign_in'
login_manager.init_app(app)

from . import views
from . import websockets

```

We need to configure where the database is located in a configuration parameter so our SQLAlchemy database loads properly. Make sure you have PostgreSQL installed on your system. Here are some handy guides and downloads to install PostgreSQL on [Ubuntu Linux](#), [Windows](#) and [Mac OS X](#).

Our new configuration variable will reside in our `config.py` file as highlighted below.

```
import os
```

```

# General Flask app settings
DEBUG = os.environ.get('DEBUG', None)
SECRET_KEY = os.environ.get('SECRET_KEY', None)

SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL', None)

# Redis connection
REDIS_SERVER = os.environ.get('REDIS_SERVER', None)
REDIS_PORT = os.environ.get('REDIS_PORT', None)
REDIS_DB = os.environ.get('REDIS_DB', None)

# Twilio API credentials
TWILIO_ACCOUNT_SID = os.environ.get('TWILIO_ACCOUNT_SID', None)
TWILIO_AUTH_TOKEN = os.environ.get('TWILIO_AUTH_TOKEN', None)
TWILIO_NUMBER = os.environ.get('TWILIO_NUMBER', None)

```

In our [srcinal Choose Your Own Adventure Presentations](#) blog post we set the other environment variables listed in the above `config.py` file. Set an environment variable for `DATABASE_URI` that will be loaded into `SQLALCHEMY_DATABASE_URI`. The value for this variable will be a URI that points to your running PostgreSQL database. For example, in my local development environment the `DATABASE_URI` is set to `postgresql://matt:password123@localhost/cyoa`. And no, that's not the actual password value I use on my development environment.

We'll need a form to handle data coming from a wizard. In our application the Flask-WTF library is going to handle form generation and input sanitization. Create a new file named `forms.py` in your `cyoa` subdirectory.

```

from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField, \
    DateField, IntegerField
from wtforms.validators import Required, Length, Regexp, EqualTo
from wtforms import ValidationError
from .models import Wizard

class LoginForm(Form):
    wizard_name = StringField('Wizard Name',
        validators=[Required(), Length(1, 32)])

```

```

password = PasswordField('Password', validators=[Required(),
                                                Length(1, 32)])

def validate(self):
    if not Form.validate(self):
        return False
    user = Wizard.query.filter_by(wizard_name=self.
                                   wizard_name.data).first()
    if user is not None and not user.verify_password(self.password.data):
        self.password.errors.append('Incorrect password.')
        return False
    return True

```

The above code creates a Python representation of the sign-in page form. We add a custom validator within the validate function to ensure the password entered by the user matches what's stored in the database. If the form input passes the validation function then True is returned and the form processing continues along in `views.py` file.

Now in `cya/cya/views.py` we need a couple of functions to handle signing wizards in and out of their special part of the application.

```

import cgi
from flask import render_template, abort, request
from flask import redirect, url_for

from flask.ext.login import login_user, logout_user, login_required, \
    current_user

from jinja2 import TemplateNotFound
from twilio import twiml
from twilio.rest import TwilioRestClient

from .config import TWILIO_NUMBER
from .forms import LoginForm
from .models import Wizard

from . import app, redis_db, socketio

from . import login_manager

```

```

client = TwilioRestClient()

@login_manager.user_loader
def load_user(userid):
    return Wizard.query.get(int(userid))

@app.route('/<presentation_name>', methods=['GET'])
def landing(presentation_name):
    try:
        return render_template(presentation_name + '.html')
    except TemplateNotFound:
        abort(404)

@app.route('/cyoa/twilio/webhook/', methods=['POST'])
def twilio_callback():
    to = request.form.get('To', '')
    from_ = request.form.get('From', '')
    message = request.form.get('Body', '').lower()
    if to == TWILIO_NUMBER:
        redis_db.incr(cgi.escape(message))
        socketio.emit('msg', {'div': cgi.escape(message),
                             'val': redis_db.get(message)},
                      namespace='/cyoa')
    resp = twiml.Response()
    resp.message("Thanks for your vote!")
    return str(resp)

@app.route('/wizard/', methods=['GET', 'POST'])
def sign_in():

    form = LoginForm()

    if form.validate_on_submit():

        wizard = Wizard.query.filter_by(wizard_name=
                                         form.wizard_name.data).first()

        if wizard is not None and wizard.verify_password(form.password.data):

```

```

    login_user(wizard)

    return redirect(url_for('wizard_landing'))

    return render_template('wizard/sign_in.html', form=form, no_nav=True)

}

@app.route('/sign-out/')
@login_required
def sign_out():

    logout_user()

    return redirect(url_for('sign_in'))

}

@app.route('/wizard/presentations/')

@login_required
def wizard_landing():

    return render_template('wizard/presentations.html')

```

What we're doing in the code above is including our new login manager we created in the `cyoa/cyoa/__init__.py` file. We then implement `login_manager` callback function named `load_user` which reloads a `wizard` object from the user ID stored in the session.

There are three additional functions:

- `sign_in` authenticate a user as a valid wizard to get into the app
- `sign_out` deauthenticate a logged in wizard's session
- `wizard_landing` a simple stub page only accessible to wizards to prove the authentication is working properly

We have the code to run the app but there are no templates to render yet. Let's create those now so we can test this thing out. Under the `cyoa/cyoa/templates` folder create a new file named `base.html` with the following template code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="">
    <meta name="author" content="Matt Makai">
    <title>{% block title %}{% endblock %}CYOA Presentations</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/
bootstrap.min.css">
    <link rel="stylesheet" href="/static/css/wizard.css">
    {% block css_head %}{% endblock %}
    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js">
      </script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js">
      </script>
    <![endif]-->
  </head>
  <body{% block body_class %}{% endblock %}>
    {% block nav %}{% endblock %}
    {% block content %}{% endblock %}
    {% block js_body %}{% endblock %}
  </body>
</html>
```

The above template serves as a base HTML template file that other templates can extend. Our sign-in page and our wizard landing page will both extend `base.html`.

You'll notice we have a new `wizard.css` file included in our `base.html` file. There are a few extra styles and a background image we will include in our app. Rather than having you type out all that code you'll want to download the file [cyoa-tutorial-step-4.tar.gz](#) and extract it under the `cyoa/cyoa/static` directory so those files can be served up when

we run our application.

`base.html` eliminates the boilerplate code we don't want to write in every template, but we can go even further with our [Don't Repeat Yourself](#)(DRY) principle by using a helper to render forms. Make a subdirectory named `partials` under the `yoa/cyoa/templates`/directory. Within `partials` construct a file named `formhelpers.html` and put the following template code inside:

```
{% macro render_field(field) %}
  <dt style="text-align: left;" class="admin-field">{{ field.label }}</dt>
  {% if field.errors %}
    <ul class="errors">
      {% for error in field.errors %}
        <li style="color: red;">{{ error }}</li>
      {% endfor %}
    </ul>
  {% endif %}
  <dd>{{ field(class_="form-control", **kwargs)|safe }}</dd>
{% endmacro %}
```

Every time we render a form field in future templates we can include the above file and it will insert the boilerplate code for a single field into the template.

Next, create a new directory in your project named `yoa/cyoa/templates/wizard`. Within the `cyoa/cyoa/templates/wizard` directory create a new file named `sign_in.html` with the following template:

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

{% block css_head %}
  <link href="/static/css/signin.css" rel="stylesheet">
{% endblock %}

{% block body_class %} class="wizard-bg" {% endblock %}

{% block content %}
<div class="container sign-in">
  <div class="col-md-12">
    <div id="login">
      <h1>Wizards Only!</h1>
```

```

{% from "partials/_formhelpers.html" import render_field %}

<form method="post" action="{{ url_for('sign_in') }}"
      id="login-form">
  {{ form.csrf_token }}
  {{ render_field(form.wizard_name, required=True) }}
  <br>
  {{ render_field(form.password, required=True) }}
  <div style="margin-top: 10px; text-align: right;">
    <input class="btn btn-primary"
           type="submit" value="Sign in" />
  </div>
</form>
</div>
</div>
</div>
</div>
<footer class="footer">
  <div class="container">
    <p class="text-muted pull-right">
      Photo by <a href="http://photos.jdhancock.com/photo/2012-04-16-000233-a-spot-of-magic.html">JD Hancock</a>
    </p>
  </div>
</footer>
{% endblock %}

```

The template renders a page with a nice wizard background image for some extra flavor. We're close to running our app, but we need a page to render once the wizard gets past the sign-in page.

Create a new file named `cyoa/cyoa/templates/wizard/presentations.html` that will store a stub page that we'll expand upon in part 2 of this tutorial.

```

{% extends "base.html" %}

{% block content %}
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <h1>Wizards Only Stuff Here</h1>
      <a href="{{ url_for('sign_out') }}>Sign out</a>
    </div>
  </div>
</div>

```

```

</div>
</div>
{% endblock %}

With our application code and templates in place we just have one more file to
update so we can run the application. Change the existing manage.py file with the
following highlighted lines.

from gevent import monkey
monkey.patch_all()

import os
import redis

from cyoa import app, redis_db, socketio

from cyoa import db
from cyoa.models import Wizard

from flask.ext.script import Manager, Shell

manager = Manager(app)

def make_shell_context():

    return dict(app=app, redis_db=redis_db, db=db, Wizard=Wizard)

manager.add_command("shell", Shell(make_context=make_shell_context))

@manager.command
def syncdb():
    db.create_all()

@manager.command
def runserver():
    socketio.run(app, "0.0.0.0", port=5001)

```

```

@manager.command
def clear_redis():
    redis_cli = redis.StrictRedis(host='localhost', port='6379', db='0')
    redis_cli.delete('left')
    redis_cli.delete('right')

if __name__ == '__main__':
    manager.run()

```

We're itching to get this application running but we need to make sure our database is created along with our `Wizard` table. On the command line create PostgreSQL database using this command:

```
createdb cyoa
```

With our updated `manage.py` script we can create the database tables necessary for storing wizards.

```
python manage.py syncdb
```

Also we need to populate our first authorized wizard. Start up the shell and enter the following code to create and save a wizard. You can modify the credentials as you see fit.

```

python manage.py shell
>>> db.session.add(Wizard('gandalf', 'thegrey'))
>>> db.session.commit()
>>> exit()

```

Finally, time to test out our new wizard sign-in screen! Within the base directory of our application run the built-in server using the following command:

```
python manage.py runserver
```

Head to <http://localhost:5001/wizard/> in a web browser to bring up our application. Now we can authenticate as a wizard with the name and password we just added to the database.

After entering our credentials and hitting the Sign in button we'll see the following landing page.



## Wizards Only Stuff Here

[Sign out](#)

Click the Sign out button to test out the `sign_out` function and ensure that we eliminate any trace of our wizard-ness to the application.

## Wizards Only Engaged

Our Wizards Only sign-in page is complete, but there's more work to be done. In part two of the Wizard Mode tutorial, we'll enable better control over presentations by building out our wizard pages.

# Choose Your Own Adventure Presentations: Wizard Mode Part 2 of 3

In the first part of our Choose Your Own Adventure Presentations Wizard Mode tutorial we set up the necessary code for separating authorized wizards from non-wizards. However, logging in and out of an application isn't very magical. It's time to write some new spells in Wizards Only mode to allow us to deftly manipulate our presentations.

## Even Wizards Need Web Forms

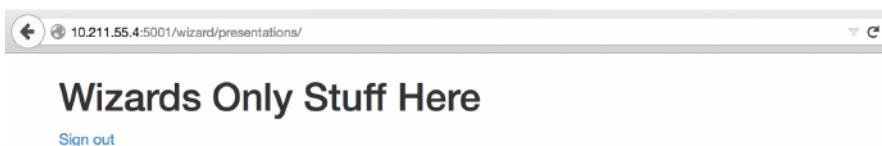
The Wizards Only mode interface we're creating throughout this post will grant us the ability to see which presentations are available as well as create and edit metadata, such as whether a presentation is visible or invisible to non-wizards.

A screenshot of a web application interface titled "Wizards Only". The top navigation bar has links for "Presentations" and "Sign out". The main content area contains four input fields: "Presentation name" (value: PyCon 2015), "File name" (value: pycon-2015.html), "URL slug" (value: pycon-2015), and "Is Visible" (checkbox checked). Below these fields is a green "Save Presentation" button.

Note that if you make a typo somewhere along the way in this section, you can compare your version with the [tutorial-step-5 tag](#). Time to get coding and create our Wizards Only mode.

## A New Wizards' Landing Page

The landing page we created in part 1 is just a stub that's not worthy of wizards who access our application.



To make the Wizards Only mode functional we'll build a new landing page that lists every presentation we've created through this user interface.

Start by opening `cyoa/cyoa/views.py` and delete the placeholder `wizard_landing` function.

```
@app.route('/wizard/presentations/')
@login_required
def wizard_landing():
    return render_template('wizard/presentations.html')
```

Next, replace what we just deleted in `cyoa/cyoa/views.py` by creating a new file named `cyoa/cyoa/wizard_views.py` and populating it with the following code.

```
from flask import render_template, redirect, url_for
from flask.ext.login import login_required

from . import app, db

@app.route('/wizard/presentations/')
@login_required
def wizard_list_presentations():
    presentations = []
    return render_template('wizard/presentations.html',
                           presentations=presentations)

@app.route('/wizard/presentation/', methods=['GET', 'POST'])
@login_required
def wizard_new_presentation():
    pass

@app.route('/wizard/presentation/<int:id>', methods=['GET', 'POST'])
@login_required
def wizard_edit_presentation(id):
    pass

@app.route('/wizard/presentation/<int:pres_id>/decisions/')
@login_required
def wizard_list_presentation_decisions(pres_id):
    pass
```

```

@app.route('/wizard/presentation/<int:pres_id>/decision/',
           methods=['GET', 'POST'])
@login_required
def wizard_new_decision(pres_id):
    pass

@app.route('/wizard/presentation/<int:presentation_id>/decision/<int:decision_id>/', methods=['GET', 'POST'])
@login_required
def wizard_edit_decision(presentation_id, decision_id):
    pass

@app.route('/wizard/presentation/<int:pres_id>/decision/<int:decision_id>/delete/')
@login_required
def wizard_delete_decision(pres_id, decision_id):
    pass

```

With the exception of the `wizard_list_presentations` function, every function above with the `pass` keyword in its body is just a stub for now. We'll flesh out those functions with code throughout the remainder of this post and also later in part 3 of the tutorial. For now we need them stubbed because otherwise the `url_for` function in our redirects and templates will not be able to look up the appropriate URL paths.

Go back to the `cyoa/cyoa/views.py` file. Edit the return redirect line shown below so it calls the new `wizard_list_presentations` function instead of `wizard_landing`

```

@app.route('/wizard/', methods=['GET', 'POST'])
def sign_in():
    form = LoginForm()
    if form.validate_on_submit():
        wizard = Wizard.query.filter_by(wizard_name=
                                         form.wizard_name.data).first()
        if wizard is not None and wizard.verify_password(form.password.data):
            login_user(wizard)

    return redirect(url_for('wizard_list_presentations'))

return render_template('wizard/sign_in.html', form=form, no_nav=True)

```

Our Flask application needs to access the new `wizard_views.py` functions. Add this single line to the end of the `cyoa/cyoa/__init__.py` file to make that happen:

```
from . import wizard_views
```

The templates for our new landing page don't exist yet, so let's create them now.

Create a new file `cyoa/cyoa/templates/nav.html` and insert the HTML template markup below.

```
<div class="container">
  <div class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand" href="{{ url_for('wizard_list_presentations') }}>Wizards Only</a>
      </div>
      <div>
        <ul class="nav navbar-nav">
          <li><a href="{{ url_for('wizard_list_presentations') }}>Presentations</a></li>
        </ul>
        <ul class="nav navbar-nav navbar-right">
          <li><a href="{{ url_for('sign_out') }}>Sign out</a></li>
        </ul>
      </div>
    </div>
  </div>
</div>
```

The above template file is a navigation bar that will be included on logged in Wizard Only pages. You'll see the template tag `{% include "nav.html" %}` in every template that needs to display the navigation bar at the top of the webpage.

Next up we need to modify the temporary markup in the landing page template file so it displays the presentations we will create through the Wizards Only user interface. Replace the temporary code found in `cyoa/cyoa/templates/wizard/presentations.html` with the following HTML template markup.

```
{% extends "base.html" %}
{% block nav %}
  {% include "nav.html" %}
```

```

{% endblock %}

{% block content %}


# Presentations


{% if not presentations %}
    No presentations found.
    >Create your first one</a>.
{% else %}
    

| Name                                                                          | Is Visible?        | Web browser voting?                                                                                          |
|-------------------------------------------------------------------------------|--------------------|--------------------------------------------------------------------------------------------------------------|
| <a href="{{ url\_for\('wizard\_edit\_presentation', id=p.id\) }}">{{ p.name }}</a> | {{ p.is\_visible }} | <a href="{{ url\_for\('wizard\_list\_presentation\_decisions', pres\_id=p.id\) }}">&gt;Manage choices&lt;/a&gt;</a> |


{% endif %}



New Presentation


```

```
</div>
</div>
{% endblock %}
```

In the above markup we check the presentations object passed into the template to determine if one or more presentations exist. If not, Flask renders the template with a “No presentations found.” message and a link to create the first presentation. If one or more presentation objects do exist, a table is rendered with the name of the presentation, whether it’s visible to non-wizard users and whether or not we’ve enabled web browser voting (which we will code in part 3).

Time to test out the current state of our application to make sure it’s working properly.

Make sure your virtualenv is activated and environment variables are set as we established in part 1 of the tutorial. From the base directory of our project, start the dev server with the `python manage.py runserver` command.

```
(cyoa)$ python manage.py runserver
 * Running on http://0.0.0:5001/
 * Restarting with stat
```

If your development server does not start up properly, make sure you’ve executed `pip`

`install -r requirements.txt` to have all the dependencies the server requires.

Occasionally there are issues installing the required gevent library the first time the dependencies are obtained via `pip`.

Open <http://localhost:5001/wizard/> in your web browser. You should see the unchanged Wizards Only sign in page. Log in with your existing Wizard credentials created in part 1. The suggested credentials for part 1 were “gandalf” for the wizard name and “thegrey” for the password.

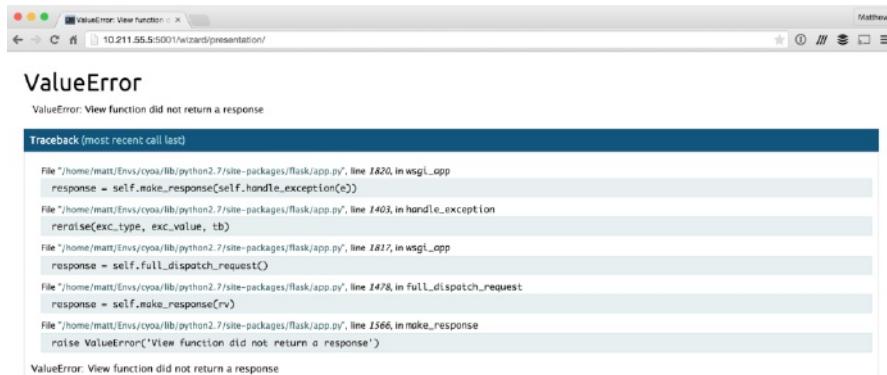
When you get into the application, the `presentations.html` template combined with our existing `base.html` and new `nav.html` will create a new landing screen that looks like this:

## Presentations

No presentations found. Create your first one.

New Presentation

However, we haven't written any code to power the "Create your first one" link and "New Presentation" buttons yet. If we click on the "New Presentation" button, we'll get a **ValueError** like we see in the screenshot below because that view does not return a response.



The screenshot shows a browser window with the URL `10.211.55.5:5001/wizard/presentation/`. The title bar says "ValueError: View function did not return a response". The main content area displays a Python stack trace:

```

ValueError: View function did not return a response

Traceback (most recent call last):
  File "/home/matt/Env/cyou/lib/python2.7/site-packages/flask/app.py", line 1820, in wsgi_app
    response = self.make_response(self.handle_exception(e))
  File "/home/matt/Env/cyou/lib/python2.7/site-packages/flask/app.py", line 1403, in handle_exception
    reraise(exc_type, exc_value, tb)
  File "/home/matt/Env/cyou/lib/python2.7/site-packages/flask/app.py", line 1827, in wsgi_app
    response = self.full_dispatch_request()
  File "/home/matt/Env/cyou/lib/python2.7/site-packages/flask/app.py", line 1478, in full_dispatch_request
    response = self.make_response(rv)
  File "/home/matt/Env/cyou/lib/python2.7/site-packages/flask/app.py", line 1566, in make_response
    raise ValueError('View function did not return a response')
ValueError: View function did not return a response

```

Let's handle creating and editing presentations next.

## Creating and Modifying Presentation Metadata

We built a page to display all presentations to logged in Wizards, but there's currently no way to add or edit presentation metadata. Why are we using the term "metadata" instead of just saying "presentations"? This Wizards Only user interface is only used to create and edit the presentations' information in the application, not the presentation

files themselves. In other words, we're modifying the presentation metadata, not the HTML markup within the presentations. As we'll see later in the post, our application will use the metadata to look in the `cyoa/cyoa/templates/presentation` folder for a filename associated with a visible presentation.

Open up `cyoa/cyoa/models.py` and append the following code at the end of the file.

```
class Presentation(db.Model):

    """ Contains data regarding a single presentation.
    """
    __tablename__ = 'presentations'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    slug = db.Column(db.String(128), unique=True)
    filename = db.Column(db.String(256))
    is_visible = db.Column(db.Boolean, default=False)

    def __repr__(self):
        return '<Presentation %r>' % self.name
```

The above `Presentation` class is a SQLAlchemy database model. Just like with our `Wizard` model, this model maps a Python object to the database table `presentations` and allows our application to create, read, update and delete rows in the database for that table.

We also need a new form to handle the creating and editing of presentation metadata. We'll store this form in the `cyoa/cyoa/forms.py` file.

```
class PresentationForm(Form):
    name = StringField('Presentation name', validators=[Required(),
                                                       Length(1, 60)])
    filename = StringField('File name', validators=[Required(),
                                                   Length(1, 255)])
    slug = StringField('URL slug', validators=[Required(),
                                              Length(1, 255)])
    is_visible = BooleanField()
```

Now we need to tie together our new `Presentation` database model and `PresentationForm` form. In the `cyoa/cyoa/wizard_views.py` remove the pass keyword from the listed functions and replace it with the highlighted code. What we're adding below are two imports for the `Presentation` and `PresentationForm` classes we just wrote. Now that we have presentations in the database, we can query for existing presentations in the `wizard_list_presentations` function. In the `wizard_new_presentation` and `wizard_edit_presentation` functions, we're using the `PresentationForm` class to create and modify `Presentation` objects through the application's web forms.

```
from . import app, db

from .models import Presentation

from .forms import PresentationForm

@app.route('/wizard/presentations/')
@login_required
def wizard_list_presentations():

    presentations = Presentation.query.all()

    return render_template('wizard/presentations.html',
                           presentations=presentations)

@app.route('/wizard/presentation/', methods=['GET', 'POST'])
@login_required
def wizard_new_presentation():

    form = PresentationForm()

    if form.validate_on_submit():

        presentation = Presentation()
        form.populate_obj(presentation)
        db.session.add(presentation)
        db.session.commit()
```

```

        return redirect(url_for('wizard_list_presentations'))

    return render_template('wizard/presentation.html', form=form, is_new=True)

@app.route('/wizard/presentation/<int:id>/', methods=['GET', 'POST'])
@login_required
def wizard_edit_presentation(id):

    presentation = Presentation.query.get_or_404(id)

    form = PresentationForm(obj=presentation)

    if form.validate_on_submit():

        form.populate_obj(presentation)

        db.session.merge(presentation)

        db.session.commit()

        db.session.refresh(presentation)

    return render_template('wizard/presentation.html', form=form,
                           presentation=presentation)

```

In the above code, make sure you've changed the first line within the `wizard_list_presentation` function from `presentations = []` to `presentations = Presentation.query.all()`. That modification to the code allows us to pass in every presentation found in the database into the `render_template` function instead of an empty list.

Create a new file named `cyoa/cyoa/templates/wizard/presentation.html`. There's already a `presentation.html` with an 's' at the end, but this filename refers to a singular presentation. Add the following HTML template within the new file:

```

{% extends "base.html" %}

{% block nav %}
    {% include "nav.html" %}
{% endblock %}

```

```

{% block content %}


{% from "partials/_formhelpers.html" import render_field %}
{% if is_new %}
<form action="{{ url_for('wizard_new_presentation') }}"
      method="post">
{% else %}
<form action="{{ url_for('wizard_edit_presentation', id=presentation.id) }}"
      method="post">
{% endif %}
<div>
  {{ form.csrf_token }}
  {{ render_field(form.name) }}
  {{ render_field(form.filename) }}
  {{ render_field(form.slug) }}
  <dt class="admin-field">
    {{ form.is_visible.label }}
    {{ form.is_visible }}
  </dt>
</div>
<div>
  <input type="submit" class="btn btn-success btn-top-margin"
         value="Save Presentation"></input>
</div>
</form>
</div>
</div>
<% endblock %>


```

Let's give our upgraded code another spin. First, since we created a new database table we need to sync our `models.py` code with the database tables. Run the following `manage.py` command at the command line from within the project's base directory.

```
(cyoa)$ python manage.py syncdb
```

Now the tables in PostgreSQL match our updated `models.py` code. Bring up the application with the `runserver` command.

```
(cyoa)$ python manage.py runserver
```

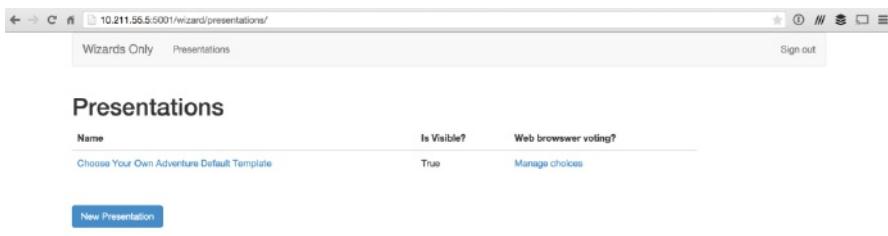
Point your web browser to <http://localhost:5001/wizard/>. You should again see the unchanged Wizards Only sign in page.



Use your wizard credentials to sign in. At the presentations landing screen after signing in, click the "New Presentation" button.

A screenshot of a web browser showing the presentations landing screen. The URL bar shows "10.211.55.5:5001/wizard/presentation/1/". The page title is "Wizards Only" and there is a "Presentations" link. On the right, there is a "Sign out" link. The main content area contains fields for "Presentation name" (set to "Choose Your Own Adventure Default Template"), "File name" (set to "cyoa.html"), "URL slug" (set to "cyoa"), and a "Is Visible" checkbox which is checked. A green "Save Presentation" button is at the bottom.

Create a presentation for the default template that's included with the project. Enter "Choose Your Own Adventure Default Template" as the presentation name, "cyoa.html" for the file name, "cyoa" for the URL slug and check the "Is Visible" box. Press the "Save Presentation" button and we'll be taken back to the presentations list screen where our new presentation is listed.



Name	Is Visible?	Web browser voting?
Choose Your Own Adventure Default Template	True	<a href="#">Manage choices</a>

We can edit existing presentations by clicking on the links within the Name column. Now we'll use this presentation information to display visible presentations to users not logged into the application.

## **Listing Available Presentations**

There have been a slew of code changes in this blog post, but let's get one more in before we wrap up that'll be useful to presentation viewers. We're going to create a page that lists all presentations that are visible to non-wizard users so you can send out a URL to anyone that wants to bring up the slides on their own.

In addition, our presentation retrieval function will look for presentation files only in the `cyoa/cyoa/templates/presentations` folder. The presentations can still be accessed from the same URL as before, but it's easier to remember where presentation files are located when there's a single folder for them.

Start by deleting the following code in `cyoa/cyoa/views.py` we will not need it any longer.

```

@app.route('/<presentation_name>', methods=['GET'])
def landing(presentation_name):
    try:
        return render_template(presentation_name + '.html')
    except TemplateNotFound:
        abort(404)

```

In its place, insert the following code.

```

@app.route('/', methods=['GET'])
def list_public_presentations():
    presentations = Presentation.query.filter_by(is_visible=True)
    return render_template('list_presentations.html',
                           presentations=presentations)

@app.route('/<slug>', methods=['GET'])
def presentation(slug):
    presentation = Presentation.query.filter_by(is_visible=True,
                                                slug=slug).first()
    if presentation:
        return render_template('/presentations/' + presentation.filename)
    abort(404)

```

The first function we wrote, `list_public_presentations`, performs a PostgreSQL database query through SQLAlchemy for all presentations with the `is_visible` field set to `True`, then passes the results to the template renderer. The second function, `presentation`, only renders the presentation if the URL slug matches an existing presentation and for that presentation the `is_visible` field is `True`. Otherwise an HTTP 404 status code is returned.

One more step in `cyoa/cyoa/views.py` Add the following line as a new import to the top of the file so that our new code can use the `Presentation` database model for the queries in the functions we just wrote:

```

from .models import Wizard

from . import app, redis_db, socketio, login_manager

from .models import Presentation

```

```

client = TwilioRestClient()

Finally create a new list template for the presentations. Call this file cyoa/cyoa/templates/list_presentations.html

{% extends "base.html" %}
{% block content %}
<div class="container">

    <div class="row">
        <div class="col-md-10">
            <h1>Available presentations</h1>
            {% for p in presentations %}
                <p><a href="{{ url_for('presentation', slug=p.slug) }}>{{ p.name }}</a></p>
            {% else %}
                No public presentations found.
            {% endfor %}
        </div>
    </div>
</div>
{% endblock %}

```

We need to move our `cyoa.html` default template into the `cyoa/yoa/templates/presentations/` folder because our `list_public_presentations()` function now looks in that folder instead of the `cyoa/yoa/templates/` folder. Create a new directory within `cyoa/cyoa/templates` called `presentations`

```
mkdir cyoa/templates/presentations/
```

Now go to the `cyoa/cyoa/templates/` directory and run the following move command.

```
mv cyoa.html presentations/
```

Be sure to store your new presentations within the `cyoa/templates/presentations/` folder from now on.

Check out the simple listing of presentations available to the audience. Go to <http://localhost:5001> and you'll see the list of visible presentations.



## All Prepared for Part 3!

We now have a working Wizards Only set of screens for managing our presentations. There's a big elephant in the room though with our current application. Each presentation has a checkbox to enable websockets-based voting to complement SMS voting. However, we haven't coded the ability for presentation watchers to vote via web browsers just yet. In part three of this series, we'll conclude by adding that new voting functionality so that our project is complete.

# Choose Your Own Adventure Presentations: Wizard Mode Part 3 of 3

## with Flask, Reveal.js and WebSockets

You've coded your way through the srcinal Choose Your Own Adventure Presentations story, the Wizards Only gatekeepeand the Web Forms trialsblog posts. Now it's time to pull out our magical wands for one more trick and complete our application-building quest.

### Voting with a wand (or smartphone)

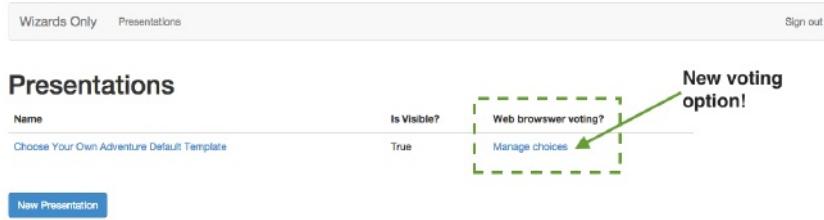
In this final tutorial we'll wrap up our Flask application with a new ability that will allow the audience to vote with a web browser on their wand... or smartphone. We will write code to keep track of audience votes from web browsers using

[websocket](#) connections. The new browser-based voting can augment our existing SMS-based voting for presentations when cell signals aren't working well.

Don that wizard's cloak one more time and get ready to write a bit more Python code as we finish out our Choose Your Own Adventure Presentations application.

## Manage Decisions View

We need to create new Wizards Only screens to set up the web browser voting when we want to enable it for presentations. Note that we don't have to enable voting via a web browser, we can continue to just use SMS for votes. However, these new screens simply give us an option to replace or augment SMS voting which is great if the cell phone service does not work well in a venue. After we write the code in this section and fire up our development server, our new admin screens should look like these screenshots below.



When we create a presentation there will be an option to manage web browser-based voting as shown above. The Manage choices link takes the wizard user to a new screen where decisions that have been created can be edited, deleted or created.

## Choose Your Own Adventures template Decisions

Decision	First story path	Second story path	Delete	View choice
into-the-cave	dragons	elves	X	into-the-cave
deeper-in-cave	further	back	X	deeper-in-cave
help-elves	help	run	X	help-elves
fight-dragon	sword	mace	X	fight-dragon
gold-treasure	keep	share	X	gold-treasure

[Add decision point](#)

Decisions can be created and saved via the simple form as seen below.

Wizards Only Presentations
Sign out

URL slug

A word for the first path. Must be lowercase. No spaces.

A word for the second path. Must be lowercase. No spaces.

[Save Decision](#)

Let's get into the new code we need to write to transform the above screenshots from images into a part of our working application. Crank open the existing `cyoa/cyoa/models.py` file in your favorite editor. We're going to add the first highlighted line to the Presentation model that represents a foreign key to the new Decision model. The Decision model will hold the slugs to the webpages that will allow someone in the audience to vote on a decision. Add the new line to Presentation and create the new Decision model as shown below.

```

class Presentation(db.Model):
    """
        Contains data regarding a single presentation.
    """
    __tablename__ = 'presentations'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    slug = db.Column(db.String(128), unique=True)
    filename = db.Column(db.String(256))
    is_visible = db.Column(db.Boolean, default=False)

    decisions = db.relationship('Decision', lazy='dynamic')

    def __repr__(self):
        return '<Presentation %r>' % self.name


class Decision(db.Model):
    """
        A branch in the storyline that an audience member can vote on.
        Must map directly into what is stored in Redis.
    """
    __tablename__ = 'choices'
    id = db.Column(db.Integer, primary_key=True)
    slug = db.Column(db.String(128))
    first_path_slug = db.Column(db.String(128))
    second_path_slug = db.Column(db.String(128))
    presentation = db.Column(db.Integer, db.ForeignKey('presentations.id'))

    def __repr__(self):
        return '<Decision %r>' % self.slug

```

The new models code above allows us to save and manipulate web browser-based decisions that can be associated with specific presentations.

Next we need to update the `cyoa/cyoa/forms.py` file with a new `DecisionForm` class. Within `cyoa/cyoa/forms.py` append the following new form class as highlighted below.

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField,
    DateField, IntegerField
from wtforms.validators import Required, Length, Regexp, EqualTo
from wtforms import ValidationError
from .models import Wizard


class LoginForm(Form):
    wizard_name = StringField('Wizard Name',
        validators=[Required(), Length(1, 32)])
    password = PasswordField('Password', validators=[Required(),
        Length(1, 32)])

    def validate(self):
        if not Form.validate(self):
            return False
        user = Wizard.query.filter_by(wizard_name=self.
            wizard_name.data).first()
        if user is not None and not user.verify_password(self.password.data):
            self.password.errors.append('Incorrect password.')
            return False
        return True


class PresentationForm(Form):
    name = StringField('Presentation name', validators=[Required(),
        Length(1, 60)])
    filename = StringField('File name', validators=[Required(),
        Length(1, 255)])
    slug = StringField('URL slug', validators=[Required(),
        Length(1, 255)])
    is_visible = BooleanField()
```

```

class DecisionForm(Form):
    slug = StringField('URL slug', validators=[Required(),
                                              Length(1, 128)])
    first_path_slug = StringField('A word for the first path. Must be '
                                  'lowercase. No spaces.',
                                  validators=[Required(), Length(1, 64),
                                              Regexp('[a-z0-9]', message=
                                                    'Choice must be lowercase '
                                                    'with no whitespace.')])
    second_path_slug = StringField('A word for the second path. Must be '
                                   'lowercase. No spaces.',
                                   validators=[Required(), Length(1, 64),
                                              Regexp('[a-z0-9]', message=
                                                    'Choice must be lowercase '
                                                    'with no whitespace.')])

```

The `DecisionForm` is used to create and edit decisions through the user interface. We've included some basic validation to ensure proper URL slugs are input by the user.

The next file that's critical for getting our web browser-based voting up and running is `cyoa/cyoa/wizard_views.py`. The changes we're going to make in this file will allow us to modify the decisions found in presentations so users can only vote in the web browser on choices we've created for them. In part two of our tutorial, we included stub functions in this `wizard_views.py` file knowing that we'd flesh them out in this post. Make sure to remove the `pass` keyword from the body of those functions and insert the highlighted code shown below.

```

from flask import render_template, redirect, url_for
from flask.ext.login import login_required

```

```

from . import app, db

from .models import Presentation, Decision
from .forms import PresentationForm, DecisionForm


@app.route('/wizard/presentations/')
@login_required
def wizard_list_presentations():
    presentations = Presentation.query.all()
    return render_template('wizard/presentations.html',
                           presentations=presentations)

@app.route('/wizard/presentation/', methods=['GET', 'POST'])
@login_required
def wizard_new_presentation():
    form = PresentationForm()
    if form.validate_on_submit():
        presentation = Presentation()
        form.populate_obj(presentation)
        db.session.add(presentation)
        db.session.commit()
    return redirect(url_for('wizard_list_presentations'))
    return render_template('wizard/presentation.html', form=form, is_new=True)

@app.route('/wizard/presentation/<int:id>/', methods=['GET', 'POST'])
@login_required
def wizard_edit_presentation(id):
    presentation = Presentation.query.get_or_404(id)
    form = PresentationForm(obj=presentation)
    if form.validate_on_submit():
        form.populate_obj(presentation)
        db.session.merge(presentation)
        db.session.commit()
        db.session.refresh(presentation)
    return render_template('wizard/presentation.html', form=form,
                           presentation=presentation)

```

```

@app.route('/wizard/presentation/<int:pres_id>/decisions/')
@login_required
def wizard_list_presentation_decisions(pres_id):

    presentation = Presentation.query.get_or_404(pres_id)
    return render_template('wizard/decisions.html', presentation=presentation,
                           decisions=presentation.decisions.all())


@app.route('/wizard/presentation/<int:pres_id>/decision/',
           methods=['GET', 'POST'])
@login_required
def wizard_new_decision(pres_id):

    form = DecisionForm()
    if form.validate_on_submit():
        decision = Decision()
        form.populate_obj(decision)
        decision.presentation = pres_id
        db.session.add(decision)
        db.session.commit()
        return redirect(url_for('wizard_list_presentation_decisions',
                               pres_id=pres_id))
    return render_template('wizard/decision.html', form=form, is_new=True,
                           presentation_id=pres_id)

@app.route('/wizard/presentation/<int:presentation_id>/decision/'
           '<int:decision_id>', methods=['GET', 'POST'])
@login_required

```

```

def wizard_edit_decision(presentation_id, decision_id):

    decision = Decision.query.get_or_404(decision_id)

    form = DecisionForm(obj=decision)

    if form.validate_on_submit():

        form.populate_obj(decision)

        decision.presentation = presentation_id

        db.session.merge(decision)

        db.session.commit()

        db.session.refresh(decision)

    return redirect(url_for('wizard_list_presentation_decisions',
                           pres_id=presentation_id))

    return render_template('wizard/decision.html', form=form,
                          decision=decision, presentation_id=presentation_id)


@app.route('/wizard/presentation/<int:pres_id>/decision/'
           '<int:decision_id>/delete/')
@login_required
def wizard_delete_decision(pres_id, decision_id):

    presentation = Presentation.query.get_or_404(pres_id)

    decision = Decision.query.get_or_404(decision_id)

    db.session.delete(decision)

    db.session.commit()

    return redirect(url_for('wizard_list_presentation_decisions',
                           pres_id=presentation.id))

```

We added code to list, create and delete decisions in the system. These are standard operations for an administrative interface that any self-respecting wizard would expect.

Alright, that's all the Python code we need at the moment for the new Wizards Only screens. Now we need to create some new templates to generate the HTML for our decision screens. Create a new file named `cyoa/cyoa/templates/wizard/decisions.html` with the following markup.

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

{% block nav %}
    {% include "nav.html" %}
{% endblock %}

{% block content %}


<div class="row">
        <div class="col-md-10">
            <h1>{{ presentation.name }} Decisions</h1>
            {% if decisions|length == 0 %}
                No web browser voting enabled for
                <em>{{ presentation.name }}</em>
                <a href="{{ url_for('wizard_new_decision', pres_id=presentation.id) }}">Add a
                decision point for this presentation</a>.
            {% else %}
                <table class="table">
                    <thead>
                        <tr>
                            <th>Decision</th>
                            <th>First story path</th>
                            <th>Second story path</th>
                            <th>Delete</th>
                            <th>View choice</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for d in decisions %}
                            <tr>


```

```

        <td><a href="{{ url_for('wizard_edit_decision',
presentation_id=presentation.id, decision_id=d.id) }}>{{ d.slug }}</a></td>
        <td>{{ d.first_path_slug }}</td>
        <td>{{ d.second_path_slug }}</td>
        <td><a href="{{ url_for('wizard_delete_decision',
pres_id=presentation.id, decision_id=d.id) }}" class="btn btn-danger">X</a></td>
        <td>{{ d.slug }}</td>
    </tr>
    {% endfor %}
</tbody>
</table>
{% endif %}
<div class="btn-top-margin">
    <a href="{{ url_for('wizard_new_decision', pres_id=presentation.id) }}" class="btn
btn-primary">Add decision point</a>
</div>
</div>
</div>
{% endblock %}

```

The above template markup loops through existing decisions and displays each one's name and first and second story paths along with options to delete or view the decision's webpage. If no decisions are returned from the database for the selected presentation, then a simple explanation will show "No web browser voting enabled for [presentation name]." There must be decision points created in the user interface for the browser-based voting to work.

There also needs to be a template file for creating new decisions and editing existing ones. Create the file `cyoa/cyoa/templates/wizard/decision.html` and insert the following markup. Make sure you've named this file without an 's' at the end of "decision" so it doesn't overwrite the previous template we just created.

```

{% extends "base.html" %}

{% block nav %}
    {% include "nav.html" %}
{% endblock %}

{% block content %}

```

```

<div class="container">
    <div class="row">
        <div class="col-md-6">
            {% from "partials/_formhelpers.html" import render_field %}
            {% if is_new %}
                <form action="{{ url_for('wizard_new_decision', pres_id=presentation_id) }}"
                      method="post">
            {% else %}
                <form action="{{ url_for('wizard_edit_decision', presentation_id=presentation_id,
decision_id=decision.id) }}" method="post">
            {% endif %}
            <div>
                {{ form.csrf_token }}
                {{ render_field(form.slug) }}
                {{ render_field(form.first_path_slug) }}
                {{ render_field(form.second_path_slug) }}
            </div>
            <div>
                <input type="submit" class="btn btn-success btn-top-margin"
                       value="Save Decision"></input>
            </div>
            </form>
        </div>
    </div>
{% endblock %}

```

With the above template we have a way to create and edit decisions. The template is generated with a form submit to either create a new decision if the "New Decision" button was clicked or edit the existing decision when modifying an existing decision.

Finally, it's time to test out our new code! Create the new Decision database table by running the following command at the base directory of our project.

```
(cyoa)$ python manage.py syncdb
```

Now run the development server with the following command as we've performed in previous blog posts.

```
(cyoa)$ python manage.py runserver
```

Head to <http://localhost:5001/wizard> to check out the current version of our application.

So far so good. Sign in using the defaults “gandalf” as a username and “thegrey” as the password that were created in part 1 of the tutorial.

If there are existing presentations in your application, click on the “Manage choices” link for one of the presentations, or create a new presentation then click the link.

Unfortunately, upon clicking “Manage choices” we will suddenly get the following error page...

### sqlalchemy.exc.ProgrammingError

```
ProgrammingError: (psycopg2.ProgrammingError) relation "choices" does not exist
LINE 2: FROM choices
[SQL: 'SELECT choices.id AS choices_id, choices.slug AS choices_slug, choices.first_path_slug AS choices_first_path_slug, choices.second_path_slug AS choices_second_path_slug, choices.presentation AS choices_presentation \nFROM choices \nWHERE %(param_1)s = choices.presentation'] [parameters: {'param_1': 1}]
```

```
Traceback (most recent call last):
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1820, in wsgi_app
    response = self.make_response(self.handle_exception(e))
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1403, in handle_exception
    reraise(exc_type, exc_value, tb)
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1817, in wsgi_app
    response = self.full_dispatch_request()
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1477, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1382, in handle_user_exception
    reraise(exc_type, exc_value, tb)
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1475, in full_dispatch_request
    rv = self.dispatch_request()
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask/app.py", line 1461, in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
  File "/home/matt/Env/cyua/lib/python2.7/site-packages/flask_login.py", line 758, in decorated_view
    return func(*args, **kwargs)
  File "/home/matt/dev/pr/cyua/wizard_views.py", line 49, in wizard_list_presentation_decisions
    decisions=presentation_decisions.all()
```

Uh oh. What’s happening here? It looks like the new foreign key relationship in thePresentation table is causing a database error. Although we created the new **decisiontable** with the **syncdb** command, it did not create the foreign key relationship column in the Presentation table.

How do we fix this problem? We need to ensure the Presentation table has the appropriate column in the database. To accomplish that we have to

- Use a library to perform a schema migration based on our updated SQLAlchemy models
- Add the column manually to the database with an ALTER TABLE statement
- Drop and recreate the database and re-sync the database tables.

For simplicity's sake, in this post we'll use the third method of dropping and recreating the database. Note that unfortunately the existing wizards and presentations in the database will be deleted when we blow away the database so we will also walk through recreating them.

Kill the development server process with **Ctrl-C** and execute the following commands to recreate the PostgreSQL database.

```
(cyoa)$ dropdb cyoa
(cyoa)$ createdb cyoa
(cyoa)$ python manage.py syncdb
```

We also need to create a new Wizard user in the database since the previous one was deleted.

```
(cyoa)$ python manage.py shell
>>> db.session.add(Wizard('gandalf', 'thegrey'))
>>> db.session.commit()
>>> exit()
```

## Finishing up browser voting

With our new Decision model in place, let's finish out this tutorial by adding the browser-based voting functionality. Open up [cyoa/cyoa/views.py](#) and update the following highlighted lines.

```
import cgi
from flask import render_template, abort, request
from flask import redirect, url_for
from flask.ext.login import login_user, logout_user, login_required,
    current_user
from jinja2 import TemplateNotFound
from twilio import twiml
from twilio.rest import TwilioRestClient

from .config import TWILIO_NUMBER
from .forms import LoginForm

from .models import Wizard, Decision

from . import app, redis_db, socketio, login_manager
from .models import Presentation

client = TwilioRestClient()

@login_manager.user_loader
def load_user(userid):
    return Wizard.query.get(int(userid))

@app.route('/', methods=['GET'])
def list_public_presentations():
    presentations = Presentation.query.filter_by(is_visible=True)
    return render_template('list_presentations.html',
        presentations=presentations)

@app.route('/<slug>', methods=['GET'])
def presentation(slug):
    presentation = Presentation.query.filter_by(is_visible=True,
                                                slug=slug).first()
    if presentation:
        return render_template('/presentations/' + presentation.filename)
    abort(404)

@app.route('/cyoa/twilio/webhook/', methods=['POST'])
```

```

def twilio_callback():
    to = request.form.get('To', '')
    from_ = request.form.get('From', '')
    message = request.form.get('Body', '').lower()
    if to == TWILIO_NUMBER:
        redis_db.incr(cgi.escape(message))
        socketio.emit('msg', {'div': cgi.escape(message),
                             'val': redis_db.get(message)},
                      namespace='/cyoa')
    resp = twiml.Response()
    resp.message("Thanks for your vote!")
    return str(resp)

@app.route('/wizard/', methods=['GET', 'POST'])
def sign_in():
    form = LoginForm()
    if form.validate_on_submit():
        wizard = Wizard.query.filter_by(wizard_name=
                                         form.wizard_name.data).first()
        if wizard is not None and wizard.verify_password(form.password.data):
            login_user(wizard)
            return redirect(url_for('wizard_list_presentations'))
    return render_template('wizard/sign_in.html', form=form, no_nav=True)

@app.route('/sign-out/')
@login_required
def sign_out():
    logout_user()
    return redirect(url_for('sign_in'))

@app.route('/<presentation_slug>/vote/<decision_slug>/', methods=['GET'])
def decision(presentation_slug, decision_slug):
    presentations = Presentation.query.filter_by(slug=presentation_slug)
    if presentations.count() > 0:
        presentation = presentations.first()

```

```

        decision = Decision.query.filter_by(presentation=presentation.id,
                                             slug=decision_slug).first()

    return render_template('decision.html', presentation=presentation,
                          decision=decision)

return render_template("404.html"), 404

```

---

```

@app.route('/<presentation_slug>/vote/<decision_slug>/<choice_slug>/',
           methods=['GET'])

def web_vote(presentation_slug, decision_slug, choice_slug):
    presentations = Presentation.query.filter_by(slug=presentation_slug)

    if presentations.count() > 0:
        presentation = presentations.first()

        decision = Decision.query.filter_by(presentation=presentation.id,
                                             slug=decision_slug).first()

        if decision:
            votes = redis_db.get(choice_slug)

            return render_template('web_vote.html', decision=decision,
                                  presentation=presentation, votes=votes,
                                  choice=choice_slug)

    return render_template("404.html"), 404

```

---

```

def broadcast_vote_count(key):

```

```

total_votes = 0

if redis_db.get(key):
    total_votes += int(redis_db.get(key))

total_votes += len(socketio.rooms['/cyoa'][key])

socketio.emit('msg', {'div': key, 'val': total_votes},
              namespace='/cyoa')

```

The above code creates two new routes for displaying voting choices and decisions built within the Wizard interface. These decision pages allow an audience member to vote by clicking one of two buttons on a page and staying on the voting page. Those votes are calculated just like SMS votes as long as the user stays on the page.

An active websocket connection increments the vote counter for that choice in Redis and when a user leaves the page the websocket connection is cleaned up and the Redis value for that choice is decremented. Note however one downside of the web browser-based voting is that the websocket connection may not be immediately recognized as closed. It can take a few seconds before the websocket is cleaned up and the vote counter decremented.

We have a couple more steps to wrap up the application's browser-voting functionality. Modify the `cyoa/cyoa/websockets.py` file by adding the highlighted code shown below.

```

from flask.ext.socketio import emit

from flask.ext.socketio import join_room, leave_room

from . import socketio

from .views import broadcast_vote_count

```

```

@socketio.on('connect', namespace='/cyoa')
def ws_connect():
    pass
    ...

@socketio.on('disconnect', namespace='/cyoa')
def ws_disconnect():
    pass
    ...

@socketio.on('join', namespace='/cyoa')
def on_join(data):
    vote = data['vote']
    join_room(vote)
    broadcast_vote_count(vote)

```

The above code handles the websockets connections and determines the decision chosen by a user.

There's one small change to the Wizard decisions page that'll make our lives easier. We want to be able to immediately view a decision after it's been created. To accomplish this task, edit the `cyoa/cyoa/templates/decisions.html` file and update the following single highlighted line.

```

{% extends "base.html" %}
{% block nav %}
    {% include "nav.html" %}
{% endblock %}

{% block content %}
<div class="container">

```

```

<div class="row">
    <div class="col-md-10">
        <h1>{{ presentation.name }} Decisions</h1>
        {% if decisions|length == 0 %}
            No web browser voting enabled for
            <em>{{ presentation.name }}</em>.
            <a href="{{ url_for('wizard_new_decision', pres_id=presentation.id) }}">Add a
            decision point for this presentation</a>.
        {% else %}
            <table class="table">
                <thead>
                    <tr>
                        <th>Decision</th>
                        <th>First story path</th>
                        <th>Second story path</th>
                        <th>Delete</th>
                        <th>View choice</th>
                    </tr>
                </thead>
                <tbody>
                    {% for d in decisions %}
                        <tr>
                            <td><a href="{{ url_for('wizard_edit_decision',
presentation_id=presentation.id, decision_id=d.id) }}>{{ d.slug }}</a></td>
                            <td>{{ d.first_path_slug }}</td>
                            <td>{{ d.second_path_slug }}</td>
                            <td><a href="{{ url_for('wizard_delete_decision',
pres_id=presentation.id, decision_id=d.id) }}" class="btn btn-danger">X</a></td>

                            <td><a href="{{ url_for('decision',
presentation_slug=presentation.slug, decision_slug=d.slug) }}" target="_blank">{{ d.slug }}</a></td>
                        </tr>
                    {% endfor %}
                </tbody>
            </table>
        {% endif %}
        <div class="btn-top-margin">
            <a href="{{ url_for('wizard_new_decision', pres_id=presentation.id) }}" class="btn
btn-primary">Add decision point</a>
        </div>
    </div>

```

```

        </div>
    </div>
</div>
{% endblock %}

```

The above one line change in `decisions.html` allows a wizard to view the decision she's created by opening a new browser window with the decision.

We're almost there - just one more template file to build! Create one more new template named `cyoa/cyoa/templates/web_vote.html` with the following contents.

```

{% extends "base.html" %}

{% block content %}


## You've chosen {{ choice }}. Stay on this page until all the votes are counted.



# <span id="vote-counter"></span> votes for {{ choice }}.


{% endblock %}

{% block js_body %}
<script type="text/javascript"
src="http://code.jquery.com/jquery-1.11.1.min.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/socket.io/0.9.16/socket.io.min.js"></script>

<script type="text/javascript">
$(document).ready(function() {
    namespace = '/cyoa';
    var websocket = io.connect('http://' + document.domain + ':' +
        location.port + namespace);

    websocket.emit('join', {'vote': '{{ choice }}');

    websocket.on('msg', function(msg) {
        var voteCounter = $('#vote-counter').html(msg.val);
    });
});
</script>

```

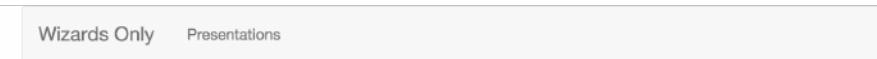
```
  });
  });
</script>
{% endblock %}
```

Let's give our application a final spin to see the web browser-based voting in action.

Run the development server with the following command as we've performed in previous blog posts.

```
(cyoa)$ python manage.py runserver
```

Head to <http://localhost:5001/wizard> to log in with our wizard account. We'll create a new presentation and add decisions to it through the new user interface to test it out.



## Presentations

No presentations found. [Create your first one.](#)

[Click to create a new presentation.](#)

Add a new presentation based with the following data or based on a presentation you've already built.

**Presentation name**

PyCon 2015

**File name**

pycon-2015.html

**URL slug**

pycon-2015

**Is Visible**

**Save Presentation**

Next create a decision point using the new code we wrote in this tutorial. This decision will allow the audience to vote with their web browser. Create a decision like the following and save it.

Wizards Only Presentations

**URL slug**

treasure-encounter

**A word for the first path. Must be lowercase. No spaces.**

take

**A word for the second path. Must be lowercase. No spaces.**

leave

**Save Decision**

Click the link in the rightmost column to view the new decision.

Wizards Only Presentations

## PyCon 2015 Decisions

Decision	First story path	Second story path	Delete	View choice
treasure-encounter	take	leave	X	<a href="#">treasure-encounter</a>

Select one of the two options on the screen and your vote will be tallied along with any other browser and SMS-based votes.

**You've chosen *take*. Stay on this page until all the votes are counted.**

## 1 vote for *take*.

You're voting for this choice via websocket just by staying on this page! This type of voting is a huge help for when cellular service doesn't work in a room or you're giving a presentation internationally where many folks don't have an active cell phone plan, such as with PyCon 2015 in Canada.

Now we've got the ability to vote both with SMS and web browsing in our Choose Your Own Adventure presentations! If there was an issue you ran into along the way that you couldn't figure out remember that there is a [tutorial-step-6 tag](#) that contains the finished code for this blog post.

### **Wizard Mode Engaged!**

Our Wizard Mode is complete. We now have far more control over presentations and a new mechanism for audience voting via web browsers in addition to SMS. With our battle-ready upgrades, we're set to use them in a live technical talk. In fact, this is the code that Kate Heddleston and I used for the talks [Choose Your Own WSGI Deployment Adventure](#) at PyCon 2015.

Let me know what new Choose Your Own Adventure presentation stories you come up with or [open a pull request](#) to improve the code base. Contact me via:

- Email: makai@twilio.com
- GitHub: Follow [makaimc](#) for repository updates

- Twitter: [@mattmakai](#)