The Definitive Guide to Masonite

Building Web Applications with Python

Christopher Pitt
Joe Mancuso

Apress®

The Definitive Guide to Masonite

Building Web Applications with Python

Christopher Pitt
Joe Mancuso

Apress®

The Definitive Guide to Masonite: Building Web Applications with Python

Christopher Pitt
Cape Town, South Africa

Joe Mancuso
Holbrook, NY, USA

ISBN-13 (pbk): 978-1-4842-5601-5
ISBN-13 (electronic): 978-1-4842-5602-2
https://doi.org/10.1007/978-1-4842-5602-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

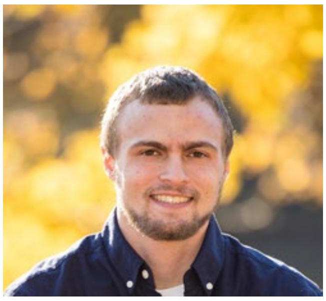
Cover designed by eStudioCalamar

Cover image designed by Raw Pixel (www.rawpixel.com)

Table of Contents

About the Authors

Christopher Pitt is a developer and writer, working at Indiefin. He usually works on application architecture, though sometimes you'll find him building compilers or robots. He is also the author of several web development books and is a contributor on various open source projects such as AdonisJs and Masonite.



Joseph Mancuso is the creator, maintainer, and BDFL for Masonite. He has been a software developer for over a decade and the CEO of Masonite X Inc, a company based around all things Masonite related. He's an avid contributor of open source, a consultant for Masonite-adopted companies, and freelance worker for companies using Masonite.

About the Technical Reviewers

Alfredo Aguirre is a senior software engineer who helps a wide range of online companies such as Mozilla, Google, R/GA, Elsewhen, and Stink Studios to transform their ideas into digital products while ensuring they perform and scale to meet the business requirements. He is based in London, UK, and can be contacted at `https://madewithbytes.com/`.

Vaibhav Mule has been programming in Python for 5 years at various organizations. He started off as an early adopter for the Masonite framework and is actively involved in maintaining Masonite and Masonite ORM.

You can learn more about him at `vaibhavmule.com`.

CHAPTER 1

Getting Started

By writing this book, we hope to teach you how to build great applications, using the Masonite framework (`https://github.com/masoniteframework/masonite`). Masonite is a modern Python framework, which includes tools and conventions aimed at making that task easier.

It's ok if you're new to Python. It's ok if you're new to Masonite. The book is written so that you can still get the most out of it. There may be some more advanced topics, along the way, but we'll do our best to make them additional and not essential to your experience.

If you've never used Python, we recommend this short course to get you familiar with the basics: `https://teamtreehouse.com/library/python-basics-3`.

The things you'll need are a computer on which to install Python, some time to read, and some time to experiment. It doesn't need to be a fancy computer, and it doesn't need to be a huge amount of time.

We've arranged the chapters so that you can learn about core concepts of the framework and build toward a functional application. You can read this book as a reference guide. You can read it as a series of tutorials. You can read it one chapter at a time, or altogether.

Thank you for taking the first step toward becoming a Masonite pro.

"Where Do I Begin?"

There are many different kinds of programming. The one you're probably most familiar with is application programming, which is where you install an application (or use an installed application) on a phone, tablet, or computer.

Come to think of it, you're also probably familiar with another kind: web site programming. These kinds of programs are used through a web browser like Chrome, Safari, or Firefox.

C. Pitt and J. Mancuso, *The Definitive Guide to Masonite*, https://doi.org/10.1007/978-1-4842-5602-2_1

Masonite sits somewhere in the middle of these two kinds of programs. Let me explain why.

Python was originally designed as a system programming language. That means it was intended to be used in short server administration scripts, to configure other software and to perform batch operations.

Over time it has become a powerful, multi-paradigm programming language. Some programming languages are mainly used for web programming, and they are used through web servers, like Apache and Nginx. Other languages take more complete control over how a web server behaves. Python is one of the latter languages.

Python web applications, and Masonite applications in particular, are usually responsible for everything from listening on a port to interpreting an HTTP request to sending an HTTP response. If something's wrong with a Masonite application, something's wrong with the whole server. With this increase in risk comes an increase in flexibility.

In addition, controlling the whole server gives us the ability to do more advanced things, like serving web sockets and interacting with external devices (like printers and assembly lines).

How Masonite Handles Releases

Before we look at code, it's important to talk about how Masonite handles releases. Big frameworks, like Masonite, change quickly. You may start a project on version 2.1, but in a few weeks version 2.2 is released. This can lead to some important questions:

- Should I upgrade?
- What does an upgrade entail?
- How often does this happen?

Let's answer these, one at a time.

Should I Upgrade?

Upgrades are a good thing, but sometimes they have trade-offs.

Functionality may be deprecated, meaning it is flagged for future removal. There may be multiple changes you need to make, so that your application will work in the new version. You may uncover bugs or things you've not tested for.

Having said all that, upgrades can also bring new features and security fixes. The security benefits alone are reason enough to take any upgrade seriously.

The best thing to do is review the upgrade guide and decide whether the cost of the upgrade is worth the benefits it brings. There's no harm in staying a few major versions back, so long as you're still using a minor version of the framework that can receive security updates (and so long as there are no obvious security issues with the version you're using).

You can find an up-to-date upgrade guide on the documentation web site: `https://docs.masoniteproject.com/upgrade-guide`.

What Does an Upgrade Entail?

This question is easily answered by reading through the upgrade guide. If you're a few versions behind, you may need to go through multiple upgrade guides to get fully up-to- date.

Masonite uses a three-part version scheme: `PARADIGM.MAJOR.MINOR`.

This means you should have little-to-no issues when upgrading from `2.1.1` to `2.1.2`. Upgrades from `2.1` to `2.2` are a bit more involved, but I've found they generally take 10 minutes or less, assuming I've not veered too far from the conventions of the framework.

In contrast to this versioning scheme, each Masonite library uses Semantic Versioning (`https://semver.org`). If you're using individual Masonite libraries, as opposed to the whole framework, then you're pretty safe upgrading from `2.1` to `2.2` without breaking changes.

How Often Does This Happen?

Masonite follows a 6-month release cycle. This means you can expect a new `MAJOR` version every 6 months. These releases aim to require less than 30 minutes to upgrade.

If they're expected to take more than that, they're shifted into a new `PARADIGM` version.

Installing the Dependencies

Masonite needs a few dependencies to work well. While we're installing those, we may also talk a bit about how best to write Python code. To begin with, let's install Python.

Installing Python and MySQL on macOS

I work on a Mac, so that's where I'd like to start things off. macOS doesn't come with a command-line package manager (like what you'd expect in Linux), so we'll need to install one.

For this section, we're going to assume you have a recent version of macOS installed, with access to the Internet.

Open Safari, and go to `https://brew.sh`. This is the home of Homebrew. It's a great package manager, which will provide us with ways to install Python 3 and a database.

There's a command, front and center. It should look something like

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

It says to download a Ruby script from that URL, executing it with the Ruby interpreter most macOS systems have installed. If you're feeling particularly suspicious, feel free to open that URL in Safari and inspect its contents.

It's good to be suspicious of sites that tell you to blindly execute scripts from the Internet. In this case, Homebrew has a reputation for security and utility. They're balancing ease of installation against the potential for that suspicion.

If you still consider this too risky, check toward the end of this section, where I recommend a more in-depth resource for setting up new Python environments.

Running this command, in terminal, will begin the process of installing Homebrew. It takes a bit of time and will ask questions along the way. One of those questions is whether you would like to install the Xcode command-line tools.

You don’t really have a choice, if you want to be able to use Homebrew. Those utilities include compilers for the code Homebrew downloads, so it won’t be able to install much without them.

When the installation is complete, you should be able to start installing dependencies through Homebrew. The ones we’re interested in are Python 3 and MySQL 5.7. Let’s install them:

```
$ brew install python3
$ brew install mysql@5.7
```

After installing MySQL, you’ll be given some instructions for starting the server. I recommend you follow these, or you won’t be able to log in or make changes to the database.

You can verify the version of Python and MySQL by running

```
$ python --version
$ mysql --version
```

You should see `Python 3.x.x` and `mysql ... 5.7.x` installed.

If that command tells you that you’re still using `Python 2.x`, then you may need to add the path, suggested at the end of the `python3` installation, to your `PATH` variable. Mine looked something like this:

```
export PATH="/usr/local/opt/python/libexec/bin:$PATH"
```

This is from `~/.zshrc`, but you should put it in `~/.profile` or `~/.bashrc`, depending on how your system is set up.

Run the `--version` commands, in a new terminal window, and you should see `Python 3.x.x` as the version.

Installing Python and MySQL on Linux

Next up, we're going to look at how to install these dependencies on Debian/Ubuntu Linux. Here, we have access to a command-line package manager, called aptitude.

You can use the following commands to install Python and MySQL:

```
$ sudo apt update
$ sudo apt install python-dev libssl-dev
$ sudo apt install mysql-server-5.7
```

If the `apt` command isn't present, you're probably using a slightly older version of Linux, where you should use `apt-get` instead.

It's a good idea to start the MySQL server, or you won't be able to log in or make changes to it:

```
$ systemctl start mysql
$ systemctl enable mysql
```

You can verify the version of Python and MySQL by running

```
$ python --version
$ mysql --version
```

You should see `Python 3.x.x` and `mysql ... 5.7.x` installed.

Editing Code

You should use the code editor, or integrated development environment, that you're most comfortable with. We recommend you use something like Visual Studio Code, as it contains just enough automated tooling to be useful, but is still fast and free.

You can download it at `https://code.visualstudio.com`.

As you open Masonite files, you'll see prompts to install Code extensions. These will give you handy tips and tell you when you have an error in your code. We recommend you install these when prompted.

Setting Up in Other Environments

If you're using macOS or Linux, these instructions should be fine for you. If you're using a different version of Linux, or Windows, you will probably need to follow a different guide for installing Python on your system.

A good place to look is the official Masonite documentation: `https://docs.masoniteproject.com`.

If you're looking for a refresher of the Python language, check out `www.apress.com/la/book/9781484200292`.

Creating a New Masonite Application

One of the tools Masonite provides, to help with project creation and maintenance, is a global command-line utility. Along with Python, the preceding instructions should also have installed a dependency management tool called Pip. We can install Masonite's command-line utility, using Pip:

```
pip install --user masonite-cli
```

Depending on how your system is set up, you may have to use a binary called `pip3` instead. If you're unsure which to use, run `which pip` and `which pip3`. These will give you hints as to where the binaries are installed, and you can pick whichever binary looks better to you.

After this command has finished executing, you should have access to Masonite's command-line utility, `craft`. You can verify this by inspecting its version:

```
craft --version
```

Now, it's time to create a new Masonite project. Navigate to where you'd like the project's code folder to reside, and run the new command:

```
craft new friday-server
```

You can substitute your own project name where you see `friday-server`. I've called mine that, for reasons that will become obvious in a bit.

You should then be prompted to navigate into the newly created folder and run an `install` command. Let's do that:

```
cd friday-server
craft install
```

This command installs the dependencies Masonite needs to run. After running this code, you should see some text telling you "key added to your .env file".

To make sure everything is working, let's run the `serve` command:

```
craft serve
```

This should tell you that the application is being served at "`http://127.0.0.1:8000`", or something similar. Open that URL in your browser, and you should be greeted with the Masonite 2.1 landing page as shown in Figure 1-1.

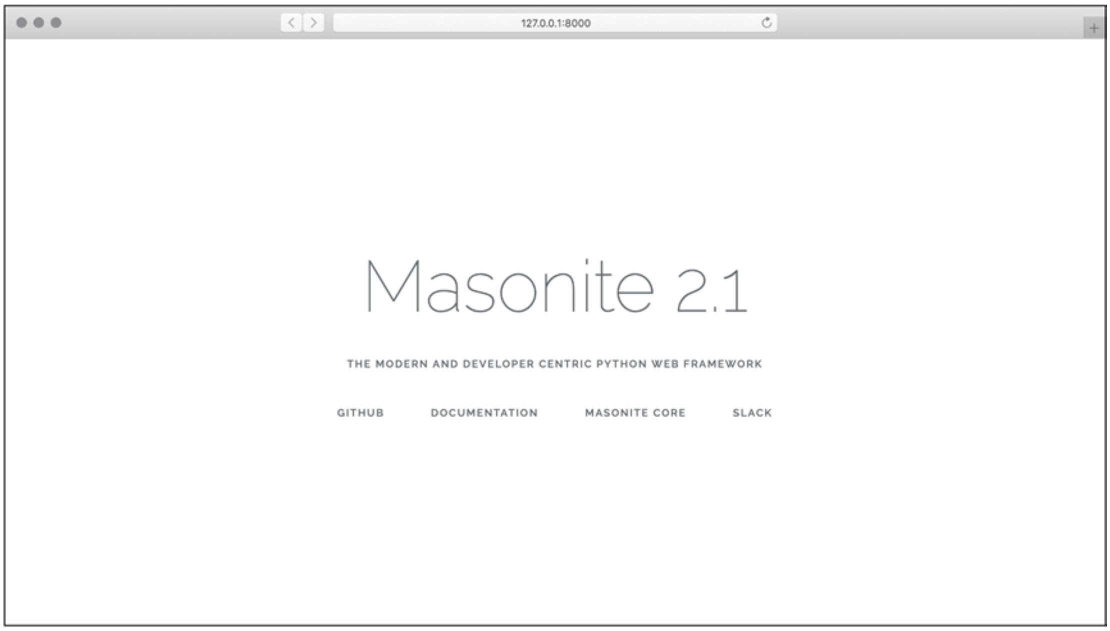
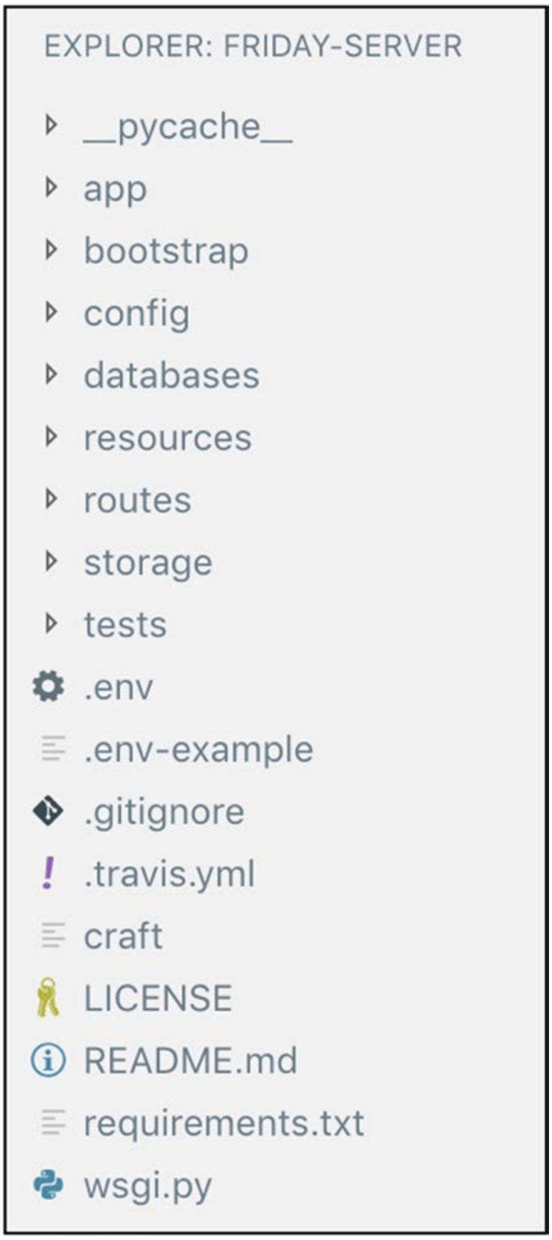


Figure 1-1. *The Masonite 2.1 landing page*

Exploring the Masonite Folder Structure

We're going to be spending a lot of time in this codebase, so a basic understanding of the folder structure (as shown in Figure 1-2) will be beneficial to us knowing where to create new files and change existing ones.



```
EXPLORER: FRIDAY-SERVER
▸ __pycache__
▸ app
▸ bootstrap
▸ config
▸ databases
▸ resources
▸ routes
▸ storage
▸ tests
  .env
  .env-example
  .gitignore
  .travis.yml
  craft
  LICENSE
  README.md
  requirements.txt
  wsgi.py
```

Figure 1-2. *The Masonite 2.1 folder structure*

Let's look at what each of these files and folders are for, without going into too much detail:

1. `app` – This folder starts off holding the parts of the application that respond to individual HTTP requests and apply blanket rules and restrictions to all requests and responses. We'll be adding lots to this folder, as we add ways for the application to respond to requests.
2. `bootstrap` – This folder hold scripts used to start the application and cache files generated during the running of the application.
3. `config` – This folder holds configuration files, which tell the Masonite application which settings to use while running.
4. `databases` – This folder holds database configuration scripts. Unlike the `config` folder's scripts, these are meant to modify an existing database, creating and modifying tables and records.
5. `resources` – This folder holds static files, like HTML templates.
6. `routes` – This folder holds files which map HTTP requests to the parts of the `app` folder which are meant to handle them. It's where we tell the application how to get from browser URLs to application files.
7. `storage` – This folder holds more static files, but generally they're the kind that we'll be putting there ourselves. Things like file uploads, Sass files, and publicly accessible files (like `favicon.ico` and `robots.txt`).
8. `tests` – This folder holds testing scripts, which we will write to make sure our application functions as intended.
9. `.env` – This file stores environment variables. These variables are likely to change between environments and are often secret values (like service keys). This file should never be committed to shared code storage locations, like GitHub. That's why the default `.gitignore` file specifically ignores `.env`.