O'REILLY®

Python Web Frameworks

Carlos de la Guardia

O'REILLY®

Python Web Frameworks

Carlos de la Guardia

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Python Web Frameworks
by Carlos de la Guardia

Printed in the United States of America.

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

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

Editor: Allyson MacDonald
Production Editor: Shiny Kalapurakkel
Copyeditor: Gillian McGarvey
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Karen Montgomery

February 2016: First Edition

Revision History for the First Edition

2016-02-12: First Release

978-1-491-93810-2

[LSI]

Table of Contents

Introduction

At the time of this writing, the web development landscape is dominated by JavaScript tools. Frameworks like ReactJS and AngularJS are very popular, and many things that were previously done on the server are handled on the client side by these frameworks. This is not limited to the client. Server-side JavaScript frameworks like NodeJS are also prominent.

Does that mean that languages like Python should throw in the towel and forget about web applications? On the contrary. Python is a very powerful language that is easy to learn and provides a fast development pace. It has many mature libraries for web-related tasks, from object-relational mapping (ORM) to web scraping. Python is also a fabulous "glue" language for making disparate technologies work together. In this era where JSON APIs and communication with multiple systems are so important, Python is a great choice for server-side web development. And it's great for full-scale web applications, too!

There are many web frameworks for Python; some provide more facilities than others, some offer a greater degree of flexibility or more extensibility. Some try to provide everything you need for a web application and require the use of very specific components, whereas others focus on giving you the bare minimum so that you can pick only the components your application needs.

Among these frameworks, there are dozens that have a significant number of users. How do newcomers to the language choose the right one for their needs? The easiest criterion would probably be popularity, and there are two or three frameworks that will easily be found doing web searches or asking around. This is far from ideal,

however, and leaves the possibility of overlooking a framework that is better suited to a developer's needs, tastes, or philosophy.

In this report, we will survey the Python web framework landscape, giving aspiring web developers a place to start their selection process for a web framework. We will look in some detail at a few of the available frameworks, as well as give pointers about how to pick one, and even how to go about creating your own.

Hopefully, this will make it easier for new developers to find what's available, and maybe give experienced Python developers an idea or two about how other web frameworks do things.

What Do Web Frameworks Do?

A *web application* is not a standalone program but part of the web "pipeline" that brings a website to a user's browser. There's much more to it than your application code working under the hood to make the web work, and having a good understanding of the other pieces of the puzzle is key to being a good web developer. In case you are new to web development or need a refresher, take a look at Appendix A to get your bearings.

When writing a web application, in addition to writing the code that does the "business logic" work, it's necessary to figure out things like which URL runs which code, plus take care of things like security, sessions, and sending back attractive and functional HTML pages. For a web service, perhaps we need a JSON rendering of the response instead of an HTML page. Or we might require both.

No matter what our application does, these are parts of it that very conceivably could be used in other, completely different applications. This is what a *web framework* is: a set of features that are common to a wide range of web applications.

Exactly which set of features a framework provides can vary a lot among frameworks. Some frameworks offer a lot of functionality, including URL routing, HTML templating systems, ORMs to interact with relational databases, security, sessions, form generation, and more. These are sometimes referred to as *full-stack frameworks*.

Other frameworks, known by many as *micro frameworks*, offer a much less varied set of features and focus on simplicity. They usually offer URL routing, templating, and not much else.

This emphasis on size (micro and full-stack) can sometimes be confusing. Are we referring to the framework's codebase? Are micro frameworks for small applications and full-stack frameworks for large applications? Also, not all frameworks easily fit into one of these categories. If a framework has lots of features but makes most of them optional, does that still count as full-stack?

From an experienced developer point of view, it could make sense to examine frameworks in terms of *decisions made*. Many features offered by frameworks, like which ORM it supports or which templating system it's bundled with, imply a decision to use that specific tool instead of other similar components.

Obviously, the more decisions made by the framework, the less decisions the developer needs to make. That means more reliance on the way the framework works, more knowledge of how its parts fit together, and more integrated behavior—all within the confines of what the web framework considers a web application. Conversely, if a developer needs to make more decisions, they'll have more work to do, but they will also have more control over their application, and can concentrate on the parts of a framework they specifically need.

Even if the framework makes these decisions, most of them are not set in stone. A developer can change these decisions, maybe by replacing certain components or libraries. The trade-off is losing some framework functionality in return for that freedom.

There are many Python web frameworks. Besides size and decisions made for the developer, many of them offer unique features or special twists on what a web appplication should do. Some developers will immediately feel attracted to some frameworks, or conclude after some analysis that one of them is better suited for the specific project they have in mind. Regardless of the chosen framework, it's always a good idea to be aware of the variety of other available frameworks so that a better choice can be made if necessary.

CHAPTER 1

Python Web Framework Landscape

There are many options for building web applications with Python. Python's powerful yet flexible nature makes it perfect for this task. It's a good idea to know what's available before going in that direction, though. Perhaps one of the many existing options will suit your needs and save you a ton of work.

To make it easier to know at a glance what frameworks are out there, the following list shows 30 web frameworks that are active and have more than 1,000 monthly downloads at the time of this writing. For each framework, the list presents the following information:

Slogan
: This is a short phrase that comes from the framework's site or documentation and attempts to convey the spirit of the framework according to its creators.

Description
: In a nutshell, what this framework is and why you should use it.

Author
: Main author, according to Python Package Index.

Website
: Official website of the framework, or code repository if no site is available.

Relative popularity
: A very crude attempt at gauging a project's popularity, by normalizing the number of monthly downloads and generating a score. Its purpose is only to give the reader a general idea about how one framework compares to another in terms of number of users. For example, Django, which is the Python framework with the largest number of downloads, has 10 stars. At the other end of the spectrum, BlueBream, which is barely above 1,000 downloads, has one star. This popularity scale should not be taken too seriously.

Python versions
: Shows the versions of Python that the framework runs on.

License
: Shows the license under which the framework is distributed.

Documentation
: This is a key part of any framework, because the more you know about how to use it, the quicker you can get started and take advantage of its features. Some people learn by example, so having tutorials and sample code can be very helpful too, both for beginners and more advanced users. For each framework, documentation is graded using a very simple scale: poor, adequate, extensive, or comprehensive. Again, this is very subjective and only meant as a simple guide to know what to expect.

Features
: A short list of what the framework's authors consider its best features.

Other resources
: This refers to resources other than web pages to get help and information for a framework, like mailing lists and IRC channels.

Persistence
: Many web applications require a storage layer of some sort, usually a database. Because of this, most web frameworks are designed to use one or more specific data persistence options.

Templating
: This is another very common feature of web frameworks. The HTML markup for an application page is usually written in a templating language.

Web Framework List

Appier

Joyful Python Web App development.

Appier is an object-oriented Python web framework built for super-fast app development. It's as lightweight as possible, but not too lightweight. It gives you the power of bigger frameworks, without the complexity.

Author

Hive Solutions Lda.

Website

http://appier.hive.pt

Relative popularity

*****

Python versions

2.6 to 3.5

License

Apache

Documentation

Adequate

Other resources

None

Persistence

MongoDB

Templating

Jinja2

Features

- REST dispatching
- JSON response encoding

- Admin interface
- i18n

Aspen

> *A Python web framework that makes the most of the filesystem. Simplates are the main attraction.*

Aspen maps your URLs directly to the filesystem. It's way simpler than regular expression routing or object traversal.

Author

Gratipay, LLC

Website

http://aspen.io

Relative popularity

******

Python versions

2.6, 2.7

License

MIT

Documentation

Adequate

Other resources

IRC

Persistence

Any

Templating

Python, Jinja2, Pystache

Features

- Simplates: code and template in same file, with structure
- JSON helpers
- Filesystem-based URL mapping

BlueBream

The Zope Web Framework.

BlueBream is an open source web application server, framework, and library created by the Zope community and formerly known as Zope 3. It is best suited for medium to large projects split into many interchangeable and reusable components.

Author

Zope Foundation and Contributors

Website

http://bluebream.zope.org

Relative popularity

*

Python versions

2.6, 2.7

License

ZPL

Documentation

Extensive

Other resources

Mailing list

Persistence

ZODB

Templating

ZPT

Features

- Built on top of Zope 3
- Full stack, but with distributed architecture
- Mature, well-tested components
- Object database

Bobo

Web application framework for the impatient.

Bobo is a lightweight framework for creating WSGI web applications. Its goal is to be easy to use and remember.

Author

Jim Fulton

Website

http://bobo.digicool.com

Relative popularity

*

Python versions

2.6 to 3.5

License

ZPL

Documentation

Extensive

Other resources

Mailing list

Persistence

Any

Templating

Any

Features

- Subroutes for multiple-step URL matching
- JSON request bodies
- Automatic response generation, based on return value

Bottle

Fast and simple WSGI framework for small web applications.

Bottle is a fast, simple, and lightweight WSGI micro web framework for Python. It is distributed as a single-file module and has no dependencies other than the Python Standard Library.

Author

Marcel Hellkamp

Website

http://bottlepy.org

Relative popularity

******

Python versions

2.6 to 3.5

License

MIT

Documentation

Extensive

Other resources

Mailing list, IRC, Twitter

Persistence

Any

Templating

Simple templates, Jinja2, Mako, Cheetah

Features

- HTTP utilities
- Single file distribution

CherryPy

A Minimalist Python Web Framework.

CherryPy allows developers to build web applications in much the same way they would build any other object-oriented Python program.

Author

CherryPy Team

Website

http://www.cherrypy.org

Relative popularity

******

Python versions

2.6 to 3.5

License

BSD

Documentation

Comprehensive

Other resources

Mailing list, IRC

Persistence

Any

Templating

Any

Features

- Authorization, sessions, static content, and more
- Configuration system
- Plugin system
- Profiling and coverage support

Clastic

> *A functional Python web framework that streamlines explicit development practices while eliminating global state.*

Clastic was created to fill the need for a minimalist web framework that does exactly what you tell it to, while eliminating common pitfalls and delays in error discovery.

Author

Mahmoud Hashemi

Website

https://github.com/mahmoud/clastic

Relative popularity

*

Python versions

2.6, 2.7

License

BSD

Documentation

Basic

Other resources

None

Persistence

Any

Templating

Any

Features

- No global state
- Proactive URL route checking
- Improved middleware paradigm

Cyclone

Facebook's Tornado on top of Twisted.

Cyclone is a web server framework for Python that implements the Tornado API as a Twisted protocol.

Author

Alexandre Fiori

Website

https://cyclone.io

Relative popularity

***

Python versions

2.6, 2.7

License

Apache

Documentation

Adequate

Other resources

None

Persistence

Twisted adbapi, redis, sqlite, mongodb

Templating

Cyclone templates

Features

- Asyncio stack
- Command-line integration

Django

The web framework for perfectionists with deadlines.

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

Author

Django Software Foundation

Website

https://djangoproject.com

Relative popularity

****

Python versions

2.6 to 3.5

License

BSD

Documentation

Comprehensive

Other resources

Mailing lists, IRC

Persistence

Django ORM

Templating

Django templates, Jinja2

Features

- Fully loaded: authentication, site maps, feeds, etc.
- Superbly documented
- Extensible admin interface
- Security-minded

Falcon

An unladen web framework for building APIs and app backends.

Falcon is a minimalist, high-performance web framework for building RESTful services and app backends with Python.

Author

Kurt Griffiths

Website

http://falconframework.org

Relative popularity

****

Python versions

2.6 to 3.5

License

Apache

Documentation

Extensive

Other resources

Mailing list, IRC

Persistence

Any

Templating

Any

Features

- Web service oriented
- Focused on performance

Fantastico

Pluggable, developer-friendly content publishing framework for Python 3 developers.

Python 3 MVC web framework with built-in capabilities for developing web services and modular web applications.

Author

Radu Viorel Cosnita

Website

https://github.com/rcosnita/fantastico/

Relative popularity

*

Python versions

3.3, 3.4, 3.5

License

MIT

Documentation

Adequate

Other resources

None

Persistence

Fantastico ORM

Templating

Any

Features

- Extensible routing engine
- ORM
- Dynamic content generation

Flask

> *Web development one drop at a time.*

A micro framework based on Werkzeug, Jinja2, and good intentions.

Author

Armin Ronacher

Website

http://flask.pocoo.org

Relative popularity

****

Python versions

2.6, 2.7, 3.3, 3.4, 3.5

License

BSD

Documentation

Comprehensive

Other resources

Mailing list, IRC

Persistence

Any

Templating

Jinja2

Features

- Built-in debugger
- RESTful request dispatching
- Allows modular applications with plugins
- Extensible

Giotto

> *Web development simplified. An MVC framework supporting Python 3.*

Giotto is a Python web framework. It encourages a functional style where model, view, and controller code is strongly decoupled.

Author

Chris Priest

Website

http://giotto.readthedocs.org

Relative popularity

**

Python versions

2.7, 3.3, 3.4, 3.5

License

Own

Documentation

Adequate

Other resources

Google group

Persistence

SQLAlchemy

Templating

Jinja2

Features

- Generic views and models
- Functional CRUD patterns
- Automatic RESTful interface
- Automatic URL routing

Grok

A smashing web framework.

Grok uses the Zope Component Architecture and builds on Zope concepts like content objects (models), views, and adapters. Its simplicity lies in using convention over configuration and sensible defaults when wiring components together.

Author

Grok Team

Website

http://grok.zope.org

Relative popularity

***

Python versions

2.6, 2.7

License

ZPL

Documentation

Extensive

Other resources

Mailing list

Persistence

ZODB

Templating

Zope page templates

Features

- Convention over configuration
- Takes advantage of full Zope toolkit
- Object-oriented database

kiss.py

MVC web framework in Python with Gevent, Jinja2, and Werkzeug.

Author

Stanislav Feldman

Website

http://stanislavfeldman.github.io/kiss.py

Relative popularity

*

Python versions

2.6, 2.7

License

Own

Documentation

Poor

Other resources

None

Persistence

Pewee

Templating

Jinja2

Features

- Integration with Gevent
- REST controllers
- Minified templates

Klein

Werkzeug + twisted.web.

Klein is a micro framework for developing production-ready web services with Python. It's built on widely used and well-tested components like Werkzeug and Twisted.

Author

Amber Brown

Website

http://klein.readthedocs.org

Relative popularity

*****

Python versions

2.6 to 3.5

License

MIT

Documentation

Adequate

Other resources

IRC

Persistence

Any

Templating

Twisted templates

Features

- Focus on web services
- Integrates Twisted concepts like deferreds

Morepath

A micro web framework with superpowers.

Morepath is a Python WSGI micro framework. It uses routing, but the routing is to models. Morepath is model-driven and flexible, which makes it expressive.

Author

Martijn Faassen

Website

http://morepath.readthedocs.org/

Relative popularity

**

Python versions

2.6 to 3.5

License

BSD

Documentation

Extensive

Other resources

Mailing list, IRC

Persistence

Any

Templating

Any

Features

- Automatic hyperkinks that don't break
- Generic UIs
- Simple, flexible permissions
- Easy to extend and override

Muffin

Web framework based on Asyncio stack.

Muffin is a fast, simple, and asyncronous web framework for Python 3.

Author

Kirill Klenov

Website

https://github.com/klen/muffin

Relative popularity

*****

Python versions

2.6 to 3.5

License

MIT

Documentation

Poor

Other resources

None

Persistence

Any

Templating

Any

Features

- Asyncio stack
- Command-line integration

Pylons

A framework to make writing web applications in Python easy.

Pylons 1.0 is a lightweight web framework emphasizing flexibility and rapid development.

Author

Ben Bangert, Philip Jenvey, James Gardner

Website

http://www.pylonsproject.org/projects/pylons-framework/

Relative popularity

****

Python versions

2.6, 2.7

License

BSD

Documentation

Extensive

Other resources

Mailing lists, IRC

Persistence

SQLAlchemy

Templating

Mako, Genshi, Jinja2

Features

- Uses existing and well-tested Python packages
- Extensible application design
- Minimalist, component-based philosophy

Pyramid

The start small, finish big, stay finished framework.

Pyramid is a general, open source, Python web application development framework. Its primary goal is to make it easier for a Python developer to create web applications.

Author

Chris McDonough, Agendaless Consulting

Website

https://trypyramid.com

Relative popularity

******

Python versions

2.6 to 3.5

License

BSD derived

Documentation

Comprehensive

Other resources

Mailing lists, IRC

Persistence

Any

Templating

Any

Features

- Powerful configuration system
- Overridable asset specifications
- Extensible templating
- Flexible view and rendering systems

Tornado

> *A Python web framework and asynchronous networking library, originally developed at FriendFeed.*

A simple web framework with asynchronous features that allow it to scale to large numbers of open connections, making it ideal for long polling.

Author

Facebook

Website

http://www.tornadoweb.org

Relative popularity

****

Python versions

2.6 to 3.5

License

Apache

Documentation

Adequate

Other resources

Mailing list, wiki

Persistence

Any

Templating

Tornado templates

Features

- Ideal for long-polling and websockets
- Can scale to tens of thousands of open connections

TurboGears

The web framework that scales with you.

TurboGears is a Python web framework based on the ObjectDispatch paradigm. It is meant to make it possible to write both small and concise applications in Minimal mode or complex applications in Full Stack mode.

Author

TurboGears Release Team

Website

http://www.turbogears.org

Relative popularity

***

Python versions

2.6, 2.7

License

MIT

Documentation

Extensive

Other resources

Mailing list, IRC, Google+

Persistence

SQLAlchemy

Templating

Genshi

Features

- From micro framework to full-stack applications
- Pluggable applications
- Widget system
- Horizontal data partitioning

Twisted

Building the engine of your Internet.

An extensible framework for Python programming, with special focus on event-based network programming and multiprotocol integration. Twisted includes twisted.web, a web application server based on the concept of resources.

Author

Glyph Lefkowitz

Website

https://twistedmatrix.com

Relative popularity

****

Python versions

2.6 to 3.5

License

MIT

Documentation

Adequate

Other resources

Mailing list, IRC

Persistence

Any

Templating

twisted.web.template

Features

- Takes advantage of Twisted networking power
- Allows "spreadable" web servers (multiple servers answer requests on same port)
- Can use any WSGI application as a resource

Uliweb

Unlimited Python web framework.

Uliweb is a full-stacked Python-based web framework. It has three main design goals: reusability, configurability, and replaceability. Its functionality revolves around these goals.

Author

Limodou

Website

http://limodou.github.io/uliweb-doc/

Relative popularity

*

Python versions

2.6, 2.7

License

BSD

Documentation

Adequate

Other resources

Mailing list

Persistence

Uliorm

Templating

Uliweb

Features

- Based on SQLAlchemy and Werkzeug
- Extensible
- Command-line tools

Watson

> *It's elementary, my dear Watson.*

A framework designed to get out of your way and let you code your application rather than spend time wrangling with the framework. It follows the "convention over configuration" ideal.

Author

Simon Coulton

Website

http://watson-framework.readthedocs.org

Relative popularity

*

Python versions

3.3, 3.4, 3.5

License

Own

Documentation

Adequate

Other resources

Mailing list

Persistence

Any

Templating

Jinja2

Features

- Event-based
- Dependency injection
- Form library

web.py

> *Think about the ideal way to write a web app. Write the code to make it happen.*

web.py is a web framework for Python that is as simple as it is powerful.

Author

Anand Chitipothu

Website

http://webpy.org

Relative popularity

*****

Python versions

2.6, 2.7

License

Public Domain

Documentation

Adequate

Other resources

Mailing list

Persistence

web.database

Templating

Templetor

Features

- Simple to use
- Own database and template libraries
- Form library

web2py

Everything in one package with no dependencies.

Free open source full-stack framework for rapid development of fast, scalable, secure, and portable database-driven web-based applications.

Author

Massimo Di Pierro

Website

http://web2py.com

Relative popularity

*

Python versions

2.6, 2.7

License

LGPL 3

Documentation

Extensive

Other resources

Mailing list

Persistence

DAL

Templating

web2py

Features

- Requires no installation and no configuration
- Web-based IDE
- Everything included; no dependencies
- Always backward-compatible

webapp2

Taking Google App Engine's webapp to the next level!

webapp2 is a lightweight Python web framework compatible with Google App Engine's webapp.

Author

Rodrigo Moraes

Website

http://webapp-improved.appspot.com/

Relative popularity

****

Python versions

2.6, 2.7

License

Apache

Documentation

Extensive

Other resources

Mailing list

Persistence

Google datastore

Templating

Jinja2, Mako

Features

- Compatible with webapp
- Better URI routing and exception handling
- Extras package with optional utilities

WebPages

A Python web framework.

This project was designed for web developers who want to do more in less time. To create a new project with *Hello World* and a database connection, you only need a few minutes.

Author

Anton Danilchenko

Website

https://github.com/webpages/webpages

Relative popularity

*

Python versions

3.3, 3.4, 3.5

License

MIT

Documentation

Poor

Other resources

Facebook

Persistence

WebPages ORM

Templating

WebPages templates

Features

- Convention over configuration
- Settings per component
- User authentication out of the box
- ORM with simplified syntax

wheezy.web

Python's fastest web framework.

A lightweight, high-performance, high-concurrency WSGI web framework with the key features to build modern, efficient web applications.

Author

Andriy Kornatskyy

Website

http://wheezyweb.readthedocs.org

Relative popularity

***

Python versions

2.6 to 3.5

License

MIT

Documentation

Adequate

Other resources

None

Persistence

Any

Templating

Jinja2, Mako, Tenjin, Wheezy

Features

- High performance
- Authentication/authorization
- Model update/validation

CHAPTER 2

Some Frameworks to Keep an Eye On

As we have seen, there are many Python web frameworks to choose from. In fact, there are too many to be able to cover every one in detail in this report. Instead, we will take a deeper look at six of the most popular. There is enough diversity here to give the reader some idea about how different frameworks work and what a web application's code looks like when using them.

For each framework, we are going to give a general description, discuss some key features, look at some sample code, and talk a bit about when it should be used. When possible, code for a simple single-file application will be shown. Quick start instructions assume Python and `pip` or `easy_install` are present on the system. It is also recommended that you use `virtualenv` (or `pyvenv` for Python 3.3+) to create an isolated environment for your application. For simplicity, the examples do not show the setup of `pip` and `virtualenv`. See Appendix A for help with any of these tools.

Django

Django is without a doubt the most popular web framework for Python at the time of this writing. Django is a high-level framework, designed to take care of most common web application needs.

Django makes a lot of decisions for you, from code layout to security. It's also very well documented, so it's very easy to get a project

off the ground quickly. There are also many third-party applications that can complement its many features nicely.

Django is very well-suited for database-driven web applications. Not only does it include its own object-relational mapping (ORM), but it can do automatic form generation based on the schemas and even helps with migrations. Once your models are defined, a rich Python API can be used to access your data.

Django also offers a dynamic administrative interface that lets authenticated users add, change, and delete objects. This makes it possible to get a nice-looking admin site up very early in the development cycle, and start populating the data and testing the models while the user-facing parts of the application are taking shape.

In addition to all this, Django has a clean and simple way of mapping URLs to code (views), and deals with things like caching, user profiles, authentication, sessions, cookies, internationalization, and more. Its templating language is simple, but it's possible to create custom tags for more advanced needs. Also, Django now supports Jinja2, so there's that option if you require a bit more powerful templating.

Quick Start

To install Django:

```
$ pip install Django
```

Unlike the other frameworks discussed in this chapter, Django is a full-stack framework, so we won't show a code listing for a *Hello World* application. While it's possible to create a Django application in a single file, this goes against the way Django is designed, and would actually require more knowledge about the various framework pieces than a complete application.

Django organizes code inside a project. A project has a configuration, or settings, plus a set of URL declarations. Since Django is intended for working with relational databases, the settings usually include database configuration information. Inside the project, there is a command-line utility, named `manage.py`, for interacting with it in various ways. To create a project:

```
$ django-admin startproject mysite
```

A project can contain one or more applications. An application is an ordinary Python package where Django looks for some things. An application can contain the database models, views, and admin site registrations that will make your models be part of the automatic admin interface. The basic idea in Django is that an application performs one defined task.

Representative Code

Django's database integration is one of its strong suits, so let's take a look at a few examples of that.

Defining models

```
from django.db import models

class Author(models.Model):
    first_name = models.CharField(max_length=70)
    last_name = models.CharField(max_length=70)

    def __str__(self):
        return self.full_name

    @property
    def full_name(self):
        return '{} {}'.format(self.first_name, self.last_name)

class Book(models.Model):
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=200)
    description = models.TextField()
    pub_date = models.DateField()

    def __str__(self):
        return self.title
```

A model contains data about one single part of your application. It will usually map to a single database table. A model is defined in a class that subclasses `django.db.models.Model`. There are several types of fields, like `CharField`, `TextField`, `DateField`, etc. A model can have as many fields as needed, which are added simply by assigning them to attributes of the model. Relationships can be expressed easily, like in the `author` field in the `Book` model above, which uses a `ForeignKey` field to model a many-to-one relationship.

In addition to fields, a model can have behaviors, or "business logic." This is done using instance methods, like `full_name` in our sample code. All models also include automatic methods, which can be overriden if desired, like the `__str__` method in the example, which gives a unicode representation for a model in Python 3.

Registering models with the admin interface

```
from django.contrib import admin
from mysite.myapp.models import Book

class BookAdmin(admin.ModelAdmin):
    list_display = ['title', 'author', 'pub_date']
    list_filter = ['pub_date']
    search_fields = ['title', 'description']

admin.site.register(Book, BookAdmin)
```

To make your models appear in the Django admin site, you need to register them. This is done by inheriting from `django.contrib.admin.ModelAdmin`, customizing the display and behavior of the admin, and registering the class, like we do in the last line of the previous example. That's all the code needed to get a polished interface for adding, changing, and removing books from your site.

Django's admin is very flexible, and has many customization hooks. For example, the `list_display` attribute takes a list of fields and displays their values in columns for each row of books, rather than just showing the result of the `__str__()` method. The hooks are not just for display purposes. You can add custom validators, or define actions that operate on one or more selected instances and perform domain specific transformations on the data.

Views

```
from django.shortcuts import render

from .models import Book

def publication_by_year(request, year):
    books = Book.objects.filter(pub_date__year=year)
    context = {'year': year, 'book_list': books}
    return render(request, 'books/by_year.html', context)
```

A view in Django can be a simple method that takes a request and zero or more URL parameters. The view is mapped to a URL using Django's URL patterns. For example, the view above might be associated with a pattern like this:

```
url(r'^books/([0-9]{4})/$, views.publication_by_year)
```

This is a simple regular expression pattern that will match any four-digit number after "books/" in the URL. Let's say it's a year. This number is passed to the view, where we use the model API to filter all existing books with this year in the publication date. Finally, the `render` method is used to generate an HTML page with the result, using a `context` object that contains any results that need to be passed to the template.

Automated Testing

Django recommends using the `unittest` module for writing tests, though any testing framework can be used. Django provides some tools to help write tests, like `TestCase` subclasses that add Django-specific assertions and testing mechanisms. It has a test client that simulates requests and lets you examine the responses.

Django's documentation has several sections dedicated to testing applications, giving detailed descriptions of the tools it provides and examples of how to test your applications.

When to Use Django

Django is very good for getting a database-driven application done really quickly. Its many parts are very well integrated and the admin site is a huge time saver for getting site administrators up and running right away.

If your data is not relational or you have fairly simple requirements, Django's features and parts can be left just sitting there, or even get in the way. In that case, a lighter framework might be better.

Flask

Flask is a micro framework. *Micro* refers to the small core of the framework, not the ability to create single-file applications. Flask basically provides routing and templating, wrapped around a few

configuration conventions. Its objective is to be flexible and allow the user to pick the tools that are best for their project. It provides many hooks for customization and extensions.

Flask curiously started as an April Fool's joke, but it's in fact a very serious framework, heavily tested and extensively documented. It features integrated unit testing support and includes a development server with a powerful debugger that lets you examine values and step through the code using the browser.

Flask is unicode-based and supports the Jinja2 templating engine, which is one of the most popular for Python web applications. Though it can be used with other template systems, Flask takes advantage of Jinja2's unique features, so it's really not advisable to do so.

Flask's routing system is very well-suited for RESTful request dispatching, which is really a fancy name for allowing specific routes for specific HTTP verbs (methods). This is very useful for building APIs and web services.

Other Flask features include sessions with secure cookies, pluggable views, and signals (for notifications and subscriptions to them). Flask also uses the concept of *blueprints* for making application components.

Quick Start

To install Flask:

```
$ pip install Flask
```

Flask "Hello World"

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

@app.route("/")
def hello():
    return "Hello World!"

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

The `Flask` class is used to create an instance of a WSGI application, passing in the name of the application's package or module. Once we

have a WSGI application object, we can use Flask's specific methods and decorators.

The `route` decorator is used to connect a view function with a URL (in this case, the root of the site). This is a very simple view that just returns a string of text.

Finally, we use the common Python idiom for executing some code when a script is called directly by the interpreter, where we call `app.run()` to start the development server.

Representative Code

Let's look at a few examples of what Flask code looks like inside real applications.

Per request connections

```
@app.before_request
def before_request():
    g.db = connect_db()

@app.teardown_request
def teardown_request(exception):
    db = getattr(g, 'db', None)
    if db is not None:
        db.close()
```

It's common to need some resources present on a per request basis such as service connections to things like redis, Salesforce, or databases. Flask provides various decorators to set this up easily. In the example above, we assume that the `connect_db` method is defined somewhere else and takes care of connecting to an already initialized database. Any function decorated with `before_request` will be called before a request, and we use that call to store the database connection in the special `g` object provided by Flask.

To make sure that the connection is closed at the end of the request, we can use the `teardown_request` decorator. Functions decorated with this are guaranteed to be executed even if an exception occurs. In fact, if this happens, the exception is passed in. In this example, we don't care if there's an exception; we just try to get the connection from `g`, and if there is one, we close it.

There's also an `after_request` decorator, which gets called with the response as a parameter and must return that or another response object.

Sessions

```
from flask import Flask, session, redirect, url_for,
escape, request

app = Flask(__name__)

@app.route('/')
def index():
    if 'username' in session:
        return 'Logged in as %s' % escape(session['username'])
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return render_template('login.html')

@app.route('/logout')
def logout():
    session.pop('username', None)
    return redirect(url_for('index'))

app.secret_key = 'A secret'
```

The `session` object allows you to store information specific to a user from one request to the next. Sessions are implemented using secure cookies, and thus need a key to be used.

The `index` view checks the session for the presence of a user name, and shows the logged-in state accordingly. The `login` view is a bit more interesting. It renders the login template if called with the GET method, and sets the session `username` variable if called with POST. The `logout` view simply removes the variable from the session, in effect logging out the user.

Views

```
@app.route('/')
def show_entries():
    cur = g.db.execute(
      'select title, text from entries order by id desc')
```

```
    entries = [dict(title=row[0], text=row[1])
      for row in cur.fetchall()]
    return render_template('show_entries.html', entries=entries)

@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('username'):
        abort(401)
    g.db.execute(
            'insert into entries (title, text) values (?, ?)',
                [request.form['title'], request.form['text']])
    g.db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

Here we show how to define views. The `route` decorator that we saw in the quickstart application is used to connect the `add_entry` method with the */add* URL. Note the use of the `methods` parameter to restrict that view to POST requests.

We examine the session to see if the user is logged in, and if not, abort the request. We assume that the request comes from a form that includes the `title` and `text` parameters, which we extract from the request to use in an insert statement. The `request` object referenced here has to be imported from `flask`, as in the sessions example.

Finally, a flash message is set up to display the change to the user, and the browser is redirected to the main `show_entries` view. This last view is simple, but it shows how to render a template, calling it with the template name and the context data required for rendering.

Automated Testing

Flask exposes the Werkzeug test client to make testing applications easier. It also provides test helpers to let tests access the request context as if they were views.

The documentation has a long section about testing applications. The examples use `unittest`, but any other testing tool can be used. Since Werkzeug is fully documented itself, there is very good information available about the test client too.

When to Use Flask

Flask can be used to write all kinds of applications, but by design it's better for small- to medium-sized systems. It is also not ideal for composing multiple applications, because of its use of global variables. It's especially good for web APIs and services. Its small core allows it to be used as "glue" code for many data backends, and it's a very powerful companion for SQLAlchemy when dealing with database-driven web applications.

Tornado

Tornado is a combination of an asynchronous networking library and a web framework. It is intended for use in applications that require long-lived connections to their users.

Tornado has its own HTTP server based on its asynchronous library. While it's possible to use the web framework part of Tornado with WSGI, to take advantage of its asynchronous nature it's necessary to use it together with the web server.

In addition to typical web framework features, Tornado has libraries and utilities to make writing asynchronous code easier. Instead of depending on callbacks, Tornado's coroutines library allows a programming style more similar to synchronous code.

Tornado includes a simple templating language. Unlike other templating languages discussed here, in Tornado templates there are no restrictions on the kind of expressions that you can use. Tornado also has the concept of *UI modules*, which are special function calls to render UI widgets that can include their own CSS and JavaScript.

Tornado also offers support for authentication and security, including secure cookies and CSRF protection. Tornado authentication includes support for third-party login systems, like Google, Facebook, and Twitter.

Quick Start

To install Tornado:

```
$ pip install tornado
```

Tornado "Hello World"

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

application = tornado.web.Application([
    (r"/", MainHandler),
])

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

First, we define a request handler, which will simply write our "Hello World" message in the response. A Tornado application usually consists of one or more handlers. The only prerequisite for defining a handler is to subclass from the `tornado.web.RequestHandler` class.

To route requests to the appropriate handlers and take care of global configuration options, Tornado uses an application object. In the example above, we can see how the application is passed the routing table, which in this case includes only one route. This route assigns the root URL of the site to the `MainHandler` created above.

Once we have an application object, we configure it to listen to port 8888 and start the asynchronous loop to serve our application. Note that there's no specific association of the application object we created and the `ioloop`, because the listen call actually creates an HTTP server behind the scenes.

Representative Code

Since Tornado's asynchronous nature is its main feature, let's see some examples of that.

Synchronous and asynchronous code

```
from tornado.httpclient import HTTPClient

def synchronous_fetch(url):
    http_client = HTTPClient()
    response = http_client.fetch(url)
    return response.body
```

```
from tornado.httpclient import AsyncHTTPClient

def asynchronous_fetch(url, callback):
    http_client = AsyncHTTPClient()
    def handle_response(response):
        callback(response.body)
    http_client.fetch(url, callback=handle_response)


from tornado import gen

@gen.coroutine
def fetch_coroutine(url):
    http_client = AsyncHTTPClient()
    response = yield http_client.fetch(url)
    raise gen.Return(response.body)
```

In these three short examples, we can see how Tornado uses asynchronous calls and how that compares with the normal, synchronous calls that we would use in a WSGI application.

In the first example, we use `tornado.HTTPClient` to fetch a URL from somewhere in the cloud. This is the regular case, and the `synchronous_fetch` call will not return until the client gets the response back.

The second example uses the `AsyncHTTPClient`. The call will return immediately after the fetch call, which is why Tornado can scale more. The fetch method is passed a callback, which is a function that will be executed when the client gets a response back. This works, but it can lead to situations where you have to chain callbacks together, which can quickly become confusing.

For this reason, coroutines are the recommended way to write asynchronous code in Tornado. Coroutines take advantage of Python generators to be able to return immediately with no callbacks. In the `fetch_coroutine` method above, the `gen.coroutine` decorator takes care of waiting without blocking for the client to finish fetching the URL, and then passes the result to the yield.

Request handlers

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("user")
```

```
class MainHandler(BaseHandler):
    def get(self):
        if not self.current_user:
            self.redirect("/login")
            return
        name = tornado.escape.xhtml_escape(self.current_user)
        self.render("hello.html", title="Welcome", name)

class LoginHandler(BaseHandler):
    def get(self):
        self.render("login.html", title="Login Form")

    def post(self):
        self.set_secure_cookie("user",
           self.get_argument("name"))
        self.redirect("/")

application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler)],
    cookie_secret="__TODO:_GENERATE_A_RANDOM_VALUE_HERE__")
```

Since request handlers are classes, you can use inheritance to define a base request handler that can have all the basic behavior needed for your application. In `BaseHandler` in the previous example, the `get_current_user` call will be available for both handlers defined in the next example.

A handler should have a method for every HTTP method that it can handle. In `MainHandler`, the `GET` method gets a look at the current user and redirects to the login handler if it is not set (remember that `get_current_user` is inherited from the base handler). If there's a user, its name is escaped before being passed to the template. The render method of a handler gets a template by name, optionally passes it some arguments, and renders it.

`LoginHandler` has both `GET` and `POST` methods. The first renders the login form, and the second sets a secure cookie with the name and redirects to the `MainHandler`. The Tornado handlers have several utility methods to help with requests. For example, the `self.get_argument` method gets a parameter from the request. The request itself can be accessed with `self.request`.

UI modules

```
class Entry(tornado.web.UIModule):
    def embedded_css(self):
        return ".entry { margin-bottom: 1em; }"

    def render(self, entry, show_comments=False):
        return self.render_string(
            "module-entry.html", entry=entry,
              show_comments=show_comments)
```

UI modules are reusable UI widgets that you can use across your application. They make it easy to design your page layouts using independent components. UI modules subclass from `tornado.web.UIModule` and must include a render method. In the example above, we define a UI module that represents a blog entry.

The render method can include arbitrary parameters, which usually will be passed on to the module template, like in the example above. A UI module can also include its own CSS and JavaScript. In our example, we use the `embedded_css` method to return some CSS to use for the entry class. There are also methods for embedding JavaScript and for pointing to CSS and JavaScript files.

Once the UI module is defined, we can call it within a template with:

```
{ % module Entry(entry, show_comments=True) % }
```

Automated Testing

Tornado offers support classes for automated testing that allow developers to test asynchronous code. It has a simple test runner, which wraps `unittest.main`. It also has a couple of test helper functions.

Tornado's test module is documented, but there is no specific tutorial or narrative section devoted to testing.

When to Use Tornado

Tornado is a bit different to the other web frameworks discussed here, in that it goes hand in hand with asynchronous networking. It's ideal to use when you need websockets, long polling, or any other kind of long-lived connections. It can also help you scale your

application to tens of thousands of open connections, provided your code is written to be asynchronous and nonblocking.

For more "regular" applications, like database-driven sites, using a WSGI framework is probably a better choice. Some of those frameworks also include a lot of features that the Tornado web framework does not have.

Bottle

Bottle is a true Python micro framework, in that it's actually distributed as a single file and has no dependencies outside of the Python standard library. It's lightweight and fast.

Bottle focuses on providing clean and simple URL routing and templating. It includes utilities for many web development needs, like access to form data, cookies, headers, and file uploads. It also includes its own HTTP server, so you don't need to set up anything else to get your application running.

Even though its core is very small and it's designed for small applications, Bottle has a plugin architecture that allows it to be expanded to fit more complex needs.

Quick Start

To install Bottle:

```
$ pip install bottle
```

Bottle "Hello World"

```
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('Hello {{name}}!', name=name)

run(host='localhost', port=8080)
```

The Bottle quick start is very simple. The `route` decorator connects the `index` method to the */hello/<name>* URL. Here, *name* is part of a dynamic route. Its value is added to the hello message using a simple string template.

Note that there is some magic going on in this example, because Bottle creates an application object the first time `route` is used. This default application object is the one that is served when the web server is run in the last line. It's also possible to create a different application object explicitly, by instantiating Bottle and assigning the result to the new application. In that case, the `run` method is passed to the application as the first argument.

Representative Code

Even though Bottle is minimalistic, it does have a few nifty features, as the following examples show.

Simple template engine

```
% name = "Bob"  # a line of python code
<p>Some plain text in between</p>
<%
  # A block of python code
  name = name.title().strip()
%>
<p>More plain text</p>

<ul>
  % for item in basket:
    <li>{{item}}</li>
  % end
</ul>
```

Bottle uses a fast, simple template language that accepts both lines and blocks of Python code. Indentation is ignored so that code can be properly aligned, but in exchange for that it's necessary to add an `end` keyword after a normally indented block, like the `for` loop above.

Using hooks

```
from bottle import hook, response, route

@hook('after_request')
def enable_cors():
    response.headers['Access-Control-Allow-Origin'] = '*'

@route('/foo')
def say_foo():
    return 'foo!'
```

```
@route('/bar')
def say_bar():
    return {'type': 'friendly', 'content': 'Hi!'}
```

Bottle supports `after_request` and `before_request` hooks, which can be used to modify the response or request. In the example above, we add a hook that will add a cross-origin resource sharing (CORS) header to the response.

Once we define the hook, it will be used in all routes. This example also shows how Bottle automatically converts a response to JSON when the callback returns a dictionary, like `say_bar` above.

Wildcard filters

```
def list_filter(config):
    ''' Matches a comma separated list of numbers. '''
    delimiter = config or ','
    regexp = r'\d+(%s\d)*' % re.escape(delimiter)

    def to_python(match):
        return map(int, match.split(delimiter))

    def to_url(numbers):
        return delimiter.join(map(str, numbers))

    return regexp, to_python, to_url

app = Bottle()

app.router.add_filter('list', list_filter)

@app.route('/follow/<ids:list>')
def follow_users(ids):
    for id in ids:
        follow(id)
```

Bottle's router supports filters to modify the value of a URL parameter before it's passed to the corresponding callback. Default filters include `:int`, `:float`, `:path`, and `:re`. You can add your own filters to the router, like in the example above.

A filter can do anything, but has to return three things: a regular expression string, a callable to convert the URL fragment to a Python object, and a callable to turn a Python object into a URL fragment. In the `list_filter` example, we will match a list of numbers in the URL and pass the converted list to the callback.

The `regexp` matches a series of one or more numbers separated by a delimiter. The `to_python` function turns them into integers, and the `to_url` function uses `str()` to turn the numbers back to strings.

Next, we explicitly create an application, and use the `add_filter` method of its router to include our filter. We can now use it in a route expression, like we do in `follow_users` at the bottom of the code.

Automated Testing

Bottle has no specific tools for testing applications. Its documentation does not have a specific section devoted to testing, though it does offer a couple of short recipes on how to do unit and functional testing.

When to Use Bottle

If you are working on a small application or a prototype, Bottle can be a good way to get started quickly and painlessly. However, Bottle does not have many advanced features that other frameworks do offer. While it's possible to add these features and build a large application with Bottle, it is designed for small applications and services, so you should definitely at least take a look at the larger frameworks if you have a complex application in mind.

Pyramid

Pyramid is a general web application development framework. It's designed for simplicity and speed. One of its main objectives is to make it possible to start small without having to have a lot of knowledge about the framework, but allow an application to grow organically as you learn more about Pyramid without sacrificing performance and organization to do so.

Pyramid focuses on the most basic web application needs: mapping URLs to code, supporting the best Python templating systems, serving static assets, and providing security features. In addition to that, it offers powerful and extensible configuration, extension, and add-on systems. With Pyramid, it's possible to override and customize core code and add-ons from the outside, making it ideal for reusable subsystems and even specialized frameworks.

Pyramid is the fruit of the merger between the *Pylons* and *repoze.bfg* web frameworks. It is developed under the *Pylons Project* brand, which sometimes causes some confusion with the old framework name.

Quick Start

To install Pyramid:

```
$ pip install pyramid
```

Pyramid "Hello World"

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response

def hello(request):
    return Response('Hello World!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('hello_world', '/')
    config.add_view(hello, route_name='hello_world')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

First, we define a view, using a function named `hello`. In Pyramid, views can be any kind of callable, including functions, classes, and even class instances. The view has to return a response (in this case, just the text "Hello World!").

To configure the application, Pyramid uses a *configurator*, to allow structured and extensible configuration. The configurator knows about several statements that can be performed at startup time, such as adding a route and a view that will be called when the route matches, as in the example above. Using a configurator might seem like an extra step in this example, but the configuration system is one of the things that make Pyramid a good fit for applications of any size, and particularly good for evolving applications.

The last step is to create an app and serve it. The configurator allows us to create the app using the `make_wsgi_app` statement. We then

pass it on to a WSGI server. Pyramid doesn't have its own server, so in the example we use the simple server included with Python.

Representative Code

Pyramid has a few unique features, like view predicates, renderers, and asset specifications. Take a look at the following examples.

View predicates

```
from pyramid.view import view_config

@view_config(route_name='blog_action',
    match_param='action=create', request_method='GET')
def show_create_page(request):
    return Response('Creating...')

@view_config(route_name='blog_action',
    match_param='action=create', request_method='POST')
def create_blog(request):
    return Response('Created.')

@view_config(route_name='blog_action',
    match_param='action=edit', request_method='GET')
def show_edit_page(request):
    return Response('Editing...')

@view_config(route_name='blog_action',
    match_param='action=edit', request_method='POST')
def edit_blog(request):
    return Response('Edited.')
```

The first thing to note in the example above is the use of the `view_config` decorator to configure views. In the Pyramid quick start example we used `config.add_view` for the same purpose, but using the decorator is a way to get the view declarations near the actual code they call.

Next, look at the different parameters passed in to the view decorator. What sets Pyramid's view configuration system apart is that unlike most frameworks, Pyramid lets developers use several pieces of information from the request to determine which views will be called by the view lookup mechanism. This is done by adding predicates to the view configuration. Think of a predicate as a True or False statement about the current request.

A view configuration declaration can have zero or more predicates, and all things being equal, the view that has more predicates that all

evaluate to True will be called by the view lookup mechanism over other views with fewer predicates. In other words, more specific view configuration declarations have preference over less specific ones.

For our example, this means that all four defined views will match the same `blog_action` route (another useful Pyramid feature), but the view lookup will look at the request method *and* at the request parameters to find the correct view to call. For example, if the request is using the POST method and there is a request parameter named "action" and that parameter has the value "edit," then the `edit_blog` view will be called.

Renderers

```
from pyramid.view import view_config

@view_config(renderer='json')
def hello_world(request):
    return {'content':'Hello!'}
```

Instead of always returning some kind of response object or assuming that a view will return a template, Pyramid uses the concept of a *renderer* for converting the result of the view to a response. When using a renderer, the view can return a dictionary and Pyramid will pass it to the renderer for generating the response.

In the example above, the view is configured with a "json" renderer, which will turn the dictionary into a response with the correct "application/json" content type header and a JSON body with the dictionary value.

Out of the box, Pyramid supports string, json, and jsonp renderers. All the supported templating add-ons for Pyramid also include a renderer. It's also very easy to create custom renderers. One benefit of this approach is simplicity, because the view code can focus on getting the correct result without worrying about generating the response and adding headers. Views that use renderers are also more easily tested, since simple unit tests will be enough to assert that the correct values are returned. Also, because rendering a response to HTML doesn't assume a specific templating engine, it's possible to easily use several engines in the same application, just by choosing the corresponding renderer (for example, "mako" or "jinja2").

Asset specifications

```
from pyramid.view import view_config

@view_config(route_name='hello_world',
    renderer='myapp:templates/hello.jinja2')
def hello(request):
    return {'message': "Hello World!"}
```

Python web applications use many types of files in addition to Python code, like images, stylesheets, HTML templates, and JavaScript code. These files are known as *assets* in Pyramid. An *asset specification* is a mechanism for referring to these files in Python code in order to get an absolute path for a resource.

In the example above, the renderer mentioned in the configuration is in fact an asset specification that refers to a Jinja2 template that can be found inside the *myapp* package. Asset specifications have two parts: a package name, and an asset name, which is a file path relative to the package directory. This tells Pyramid that the template to be used for rendering the view is the *hello.jinja2* file inside the *templates* directory in the *myapp* package. Pyramid resolves that into an absolute file path, and passes it to the Jinja2 renderer for generating the HTML.

Pyramid supports serving static assets, which use the same specifications for pointing at the assets to be served. This mechanism allows Pyramid to easily find and serve assets without having to depend on configuration globals or template hierarchies, like other frameworks do.

Asset specifications also make it possible to override any asset. This includes template files, template directories, static files, and static directories. For example, suppose the view declaration in the example is part of a reusable application that is meant to be mostly used as is, but with a different look. Pyramid allows you to reuse the entire application from another package and simply override the template, without requiring that the application be forked. All that would be needed is to add the following code to the customized application's configuration:

```
config.override_asset(
    to_override='myapp:templates/hello.jinja2',
    override_with=
      'another.package:othertemplates/anothertemplate.jinja2')
```

Automated Testing

Pyramid has its own `testing` module for supporting tests. Like other frameworks, it uses `unittest` in the documentation. Key Pyramid components, like the configurator, offer test helpers to simulate some of their functionality.

Pyramid's documentation includes a chapter about unit, integration, and functional testing. The `pyramid.testing` module is also documented, as are the various testing helpers. All official tutorials include a testing section.

When to Use Pyramid

Pyramid is designed to be used for small to large applications, so unlike some of the frameworks discussed here, size is not a criterion for deciding on its use. Pyramid has a few features that work differently from other frameworks, so being comfortable with those features is important if you want to use it.

Pyramid is very well-suited for writing extensible applications, and is a very good candidate for writing domain-specific frameworks due to its many extension points and override mechanisms.

CherryPy

CherryPy is the oldest framework discussed here. It is perhaps the original Python micro framework, as it focuses on matching a URL to some view, and leaves everything else to the developer. It does include its own web server, so it can serve static content as well. In addition, CherryPy has built-in tools for caching, encoding, sessions, authorization, and more.

CherryPy has a fine-grained configuration system, which allows several configuration settings to be modified. Global configuration is separate from application configuration, so it's possible to run several applications in one process, each with its own configuration.

CherryPy is designed to be extensible, and offers several mechanisms for extensions and hook points. Since it has its own server, these extension points include server-wide functions outside of the regular request/response cycle. This gives CherryPy an added level of extensibility.

Quick Start

To install CherryPy:

```
$ pip install cherrypy
```

CherryPy "Hello World"

```
import cherrypy

class HelloWorld(object):
    def index(self):
        return "Hello World!"
    index.exposed = True

cherrypy.quickstart(HelloWorld())
```

CherryPy applications are usually written as classes. The *views* of the application are the methods of its class, but by default only explicitly selected methods will be turned into views, or *exposed*, as CherryPy calls it. In the example, we have a single class named `HelloWorld`, with a single method in it that simply returns the text "Hello World!". The index method is explicitly exposed by setting that attribute to True.

Then, we use the `cherrypy.quickstart` call to direct CherryPy to host our application. By default, it will be accessible on *http://127.0.0.1:8080/*.

Note that unlike other frameworks, CherryPy does not associate a URL to a view using mappings or routes. By default, the `index` method goes to "/". Other exposed methods of a class are located by using the method name as a URL segment. For example, a `goodbye` method in the example above would be called by a request to *http://127.0.0.1:8080/goodbye* (if exposed, of course). CherryPy does provide other ways to handle parameters in routes.

Representative Code

Here are a couple of examples that showcase some of CherryPy's features.

REST API

```
import random
import string
```

```
import cherrypy

class StringGeneratorWebService(object):
    exposed = True

    def GET(self):
        return cherrypy.session['mystring']

    def POST(self, length=8):
        some_string = ''.join(random.sample(string.hexdigits,
            int(length)))
        cherrypy.session['mystring'] = some_string
        return some_string

if __name__ == '__main__':
    conf = {
        '/': {
            'request.dispatch':
                cherrypy.dispatch.MethodDispatcher(),
            'tools.sessions.on': True,
        }
    }
    cherrypy.quickstart(StringGeneratorWebService(), '/', conf)
```

First, we define a class. Instead of exposing every method individually, we set `exposed` to True for the whole class, which exposes all its methods. We define separate methods for each HTTP "verb." They are very simple. GET will get the latest generated string, and `POST` will generate a new one. Note the use of `cherrypy.session` to store information for the current user from request to request.

A REST API uses the request methods to decide which views to call in an application. The default CherryPy dispatcher does not know how to do that, so we need to use a different dispatcher. This is done in the configuration dictionary that will be passed to `cherrypy.quickstart`. As you can see, CherryPy already has a dispatcher for this, so we just set it and everything will work.

JSON encoding and decoding

```
class Root(object):

    @cherrypy.expose
    @cherrypy.tools.json_in()
    def decode_json(self):
        data = cherrypy.request.json

    @cherrypy.expose
```

```
    @cherrypy.tools.json_out()
    def encode_json(self):
        return {'key': 'value'}
```

These days, JavaScript is used extensively in web applications, and JSON is the standard way to get information to and from JavaScript applications. CherryPy has a simple way of encoding and decoding JSON.

In the first example, we can treat a request as JSON using the `cherrypy.tools.json_in` decorator. It attaches a "json" attribute to the request, which contains the decoded JSON.

In the second example, we use the corresponding `cherrypy.tools.json_out` decorator, which encodes the JSON and generates the correct "application/json" header.

Tools

```
import time

import cherrypy

class TimingTool(cherrypy.Tool):
    def __init__(self):
        cherrypy.Tool.__init__(self, 'before_handler',
                               self.start_timer)

    def _setup(self):
        cherrypy.Tool._setup(self)
        cherrypy.request.hooks.attach('before_finalize',
                                      self.end_timer)

    def start_timer(self):
        cherrypy.request._time = time.time()

    def end_timer(self):
        duration = time.time() - cherrypy.request._time
        cherrypy.log("Page handler took %.4f" % duration)

cherrypy.tools.timeit = TimingTool()

class Root(object):
    @cherrypy.expose
    @cherrypy.tools.timeit()
    def index(self):
        return "hello world"
```

It's very common in a web application to need some code to be executed at some point for every request. CherryPy uses *tools* for that

purpose. A tool is a callable piece of code that is attached into some specific point of the request, or *hook*. CherryPy offers several hooks, from the very beginning of the request until it is completed.

In the example, we define a tool for computing the time taken by a request from start to finish. A tool needs an `__init__` method to initialize itself, which includes attaching to a hook point. In this case, the tool is attached to the `before_handler` hook, which is processed right before the view is called. This starts the timer, saving the start time as an attribute of the request.

After a hook is initialized and attached to the request, its `_setup` method is called. Here, we use it to attach the tool to yet another hook, `before_finalize`. This is called before CherryPy formats the response to be sent to the client. Our tool will then stop the timer by calculating the difference between the start and end times, and logging the result.

To initialize the tool, we instance it and add it as an attribute of `cherrypy.tools`. We can now use it in our code by adding it as a decorator to any class method, like this: `@cherrypy.tools.timeit()`.

Automated Testing

CherryPy provides a helper class for functional tests. It is a test case class with some helpers. The documentation for this class is brief, but includes an example.

When to Use CherryPy

CherryPy is a very flexible framework, which can be used in applications of all kinds. The fact that it includes its own web server and doesn't have many dependencies makes it a good choice for shared web hosting sites.

Though it's very easy to get started with, CherryPy does leave in the developer's hands many of the tasks that some frameworks handle for them. This is flexible but might require more work. Also, some of its configuration and extensibility features need a more thorough understanding, so beginners trying to use it right away for a big undertaking might find its learning curve a bit steep.

CHAPTER 3

What's the Right Framework for You?

By this point, you know a little bit about which Python web frameworks are out there, and even have a little in-depth knowledge about some of them. Still, you might be wondering which one is best for you. This is not a question that can be answered here to your satisfaction, because every reader of this guide will have different needs. However, I can offer some general guidelines and answer a couple of frequently asked questions by people new to Python web development.

Don't Look for Absolute Bests

Recommending one framework over all of the others is not the purpose of this guide, but it's possible to tell you which frameworks not to look for. New web developers in community forums like reddit usually start looking for one of these, and even if they could find it, it would be the wrong approach to solving their specific problem.

The "Best" Framework

If you search the archives of Python web forums, you'll find that this question appears every couple of weeks. The thinking goes like this: "Hey, if I'm going to get into web development, I should use the best framework out there." The problem is that there's no one right answer to this question.

That's why this book contains a list of 30 frameworks and reviews for half a dozen of them. The reality is that many frameworks could be considered the "best," depending on your specific objectives and your preferred way of working. Picking one or two of the frameworks discussed here and trying them out to see if they are a fit for the way you like to code is a much better use of your time than trying to find an undisputed champion.

The "Fastest" Framework

It's not bad to look for speed in a framework, but making that your main concern can get you off track easily. For example, it can make you worry about deployment before you actually write a line of code. Talk about putting the cart in front of the horse!

The truth is that for many applications, the web framework will seldom be the bottleneck, and even if you really need speed, it's pointless to have that if the rest of the framework does not conform to your requirements and objectives.

The "Smallest" Framework

When you have lots of programs that do more or less the same thing, like web frameworks, it's natural to try to classify them in some way. In the Python web world, the terms *full-stack framework* and *micro framework* are the most used categories these days. The former refers to a framework with many integrated features, like sessions, authorization, authentication, templating, and database access; the latter usually means a smaller but very focused framework that concentrates on the main things a web application must do, like URL routing.

My advice would be, don't get too caught up in taxonomies. It's not necessarily logical that "small" applications require small frameworks and "big" applications require large frameworks. For example, even if you know that you'll just need a "small" application, you might need several services that micro frameworks do not usually offer, like integrated sessions and form handling. Conversely, you could be writing a "large" application, but more in terms of business logic than web requirements. Instead of worrying about size, focus on the features you need.

Start by Defining Your Goals

If there's a lesson from the previous section, it's that to find the right framework for you, it's better if you have a clear idea of what you are trying to accomplish. Define your problem first, and then look for a framework that seems well-suited for solving that problem and fits the way you think.

Also, once you pick a framework for a project, it's not as if you are somehow bound by contract to always use that framework. You may very well find that your next project requires a completely different approach, where another framework might bring more to the table.

Desirable Features

While there's no general checklist for finding the best framework for your use case, there are indeed a few things that you can look for that could simplify your work a lot. It would be wise to at least consider these points while you decide which framework to use.

Documentation

This is probably the single most important thing to look for in a framework. Good documentation lets you get up to speed quickly and start worrying about your application right away rather than trying to find out how to use the framework's features.

Bonus points if the documentation is well-written, easy to follow, and offers a variety of resources, like tutorials, cookbooks, and videos. People learn in different ways, and some resources, like tutorials, are usually much better for beginners. A discussion of advanced topics can be very useful once you start familiarizing yourself with the framework and trying more ambitious things.

Good documentation also evolves with the framework, so new releases should always cover new features; there are few things as frustrating as having to deal with outdated documentation when trying to pinpoint if a problem is in your code or in the way you are trying to use the framework.

Active Community

It's a good idea to try to connect with the community of users and developers of your framework of choice. There's no better way to gauge existing interest in the framework and get a feel for how much development work is under way to make it better. Usually, frameworks will have some mix of mailing lists, IRC channels, and web forums. For most frameworks, these communication channels will also be the official support mechanism, so it pays to find out about them.

An active community is not only a good way of finding out if a framework is "alive" and current, but it can also be a very nice complement to the documentation. Finding out how people have used a feature in real life can be as helpful as the description of the feature itself.

Reliability

The Python language evolves, new ways of working with the Web come around frequently, and the open nature of the Web means constant attention needs to be paid to security considerations. A reliable framework is one that tries to stay current with all of these events. Look for versions that mention compatibility with newer Python releases, take advantage of new libraries and ways of doing things, and offer occasional security fixes or notices.

Extensibility

Once you become proficient in the use of a framework, there usually comes a time when you need to add some functionality or feature that is not already part of it, and in a way that can be easily maintained and understood. If a framework offers well-designed and documented extension points, it will be easier to adapt to your special requirements in a way that doesn't break with version updates. Also, it will be easier to take advantage of general extensions or plugins that other framework users have created.

CHAPTER 4

Developing Your Own Framework

As the final step in our tour of Python web frameworks, we'll look briefly at how to develop your very own framework. Even if you don't plan to build one, knowing a bit about how the internals work might be a good exercise.

Why Create a Framework?

The framework list in this report has 30 frameworks, and that's just the ones that have a significant(ish) number of downloads. There are many more. Still, people keep creating frameworks, so you might some day want to do that too. In fact, one of the reasons that there are so many Python web frameworks is that it's not very hard to create one. Python has very good building blocks for this, starting with the WSGI standard and libraries like Werkzeug and WebOb that already implement all of the plumbing, which can be used as a starting point.

Sometimes, instead of a general framework, you might need one that is more tailored to your problem domain. It's much easier to start from a solid foundation, so you could consider basing your special domain framework on an existing one. Frameworks that have flexible configuration and extension mechanisms, like Pyramid or CherryPy, lend themselves very well to this approach.

Parts of a Basic WSGI Framework

If you decide to build a framework, using Werkzeug or WebOb is a good recommendation. You want to focus on the user-configurable parts, not the internals.

Routing

You will need some way to match a URL to a view. The most common way of doing this is using regular expression routes, like Django or Flask. If you want to go for this approach, you should consider using Werkzeug, which includes a routing system right out of the box. There are also some routing libraries available on PyPI that can be used with WebOb, or by themselves.

There are other ways to map URLs to views. Pyramid offers *traversal*, which treats your site's content as a tree and maps views to resources or types of resources. CherryPy uses exposed methods from regular Python classes. Maybe you can think about other approaches. Just use something that feels natural to you.

Templates

There are a couple of very popular template systems for Python, and it's highly recommended to use one of them. Jinja2 and Mako are two very good options.

Some frameworks are highly tailored to work with a chosen template system, while others leave that decision completely in the hands of the developer. Whatever your choice, you can't go wrong, as most of these template systems are proven and very stable.

Other Features

Most frameworks offer other features as part of the package. Authentication is a common need and thus is usually provided for in some way. There are many authentication systems and libraries out there. The best advice here is to try to be flexible. You never know how a user will need to authenticate their users.

Many frameworks make the assumption (perhaps less of a certainty these days) that a relational database backend will be part of a web application, and offer at least some degree of database support. If

that's your decision, simply go for SQLAlchemy integration. It's the most complete (and popular) solution for Python.

Consider offering some support for form handling and input validation, but be aware that there are many libraries that do this very well. Research your alternatives a bit before deciding to do it yourself.

If you intend for your framework to be used by other people, or at least plan to use it more than once, you need to offer a configuration system that allows your users to easily set up and deploy their applications.

Documentation

Even if your framework is only for internal use at your company, do your best to document it. In fact, if you can't document it, it is better to use one of the existing frameworks. Future maintainers of your applications will thank you.

Good documentation consists of much more than class names and methods. Some narrative documentation is very useful, and many people find that code examples speak volumes. Beginners like tutorials and less technical explanations. Use a documentation tool, like *sphinx*, to make documentation generation easier.

Framework Building Blocks

You don't have to start your framework from scratch. There are a couple of *WSGI toolkits* that take care of the basics, allowing you to concentrate on the unique features provided by your framework.

The two most popular WSGI toolkits are WebOb and Werkzeug. Many web frameworks use one of these libraries. They are time-tested and fully featured, so they provide an excellent base for developing web applications without worrying about the details of HTTP and WSGI.

WebOb (http://webob.org)

WebOb provides objects that map much of the specified behavior of HTTP, including header parsing, content negotiation, and correct handling of conditional and range requests. Pyramid is one well-known framework that uses WebOb.

Werkzeug (http://werkzeug.pocoo.org)

Werkzeug is one of the most advanced WSGI utility modules. It includes a powerful debugger, fully featured request and response objects, HTTP utilities to handle entity tags, cache control headers, HTTP dates, cookie handling, file uploads, and a powerful URL routing system. Flask, one of the most popular Python web frameworks, uses Werkzeug.

Some Useful Resources

If you do decide to write your own framework, or just want to know a bit more about how to do it, there are some excellent web resources that can show you how. Here are three of the best.

You can start with *A Do-It-Yourself Framework*, in which Ian Bicking explains what WSGI is by building a simple framework. This is a very good tutorial and a great way to find out what's inside a real framework.

Ian Bicking has a second tutorial, this time using the WebOb library. *Another Do-It-Yourself Framework* has more details about a framework's parts, and even goes line by line to explain the code.

Some people learn better using screencasts instead of written documentation. If that's your case, you'll find Chris McDonough's series of screencasts about using repoze.bfg to build a micro framework useful. Remember, repoze.bfg is the framework that became Pyramid. Even if you don't use either of these frameworks, you should still find this series very instructive. The videos are at http://bfg.repoze.org/videos.

CHAPTER 5
Summary

We have completed a very quick tour of the Python web framework world. There are many Python web frameworks and we offered a glimpse into a good number of them. We also covered six popular frameworks in detail, which hopefully will tempt you to try one or more of them on for size. Some general advice about how to pick a framework and even build your own was given, but as always, the best way to really know something is to try it yourself.

APPENDIX A

Python Web Development Fundamentals

It's All HTTP Underneath

When you are working with web applications, every operation is ultimately reduced to a series of requests and responses between the user's browser and the server hosting the application, using the HTTP protocol. While an introductory discussion of HTTP is outside the scope of this book, it's important for web developers to know how it works.

From a web application standpoint, HTTP means getting requests from a web browser for some URL that the application knows how to handle, then processing this request and sending a response back. This sounds easy, but there's lots of stuff to do.

Imagine that we are writing an application that will do all this work without depending on other libraries or frameworks. Our application needs to know how to interpret a request, decode any parameters that it contains, pass that information to the specific code in the application that is supposed to be run for that URL, get a result (preferably a nice HTML page with a beautiful design), encode that into a response, and send it back to the server. All of this would require writing code to handle each task.

Additionally, HTTP is a stateless protocol, which means no session information is kept from request to request. Our application will need some way to know if a user logs in or out, and make sure that

the correct information for each user is used where required. The browser provides some tools to keep track of that on the client side, but the application needs to do some work, and the available mechanisms are often not enough for many applications.

Our application also has to have a way to get requests from a web server, as well as sending back a response. That means more code will be needed to communicate with the web server. The actual logic in our application begins to feel rather small compared to the work required to make it work on the Web.

Web Servers and WSGI

When the Web started, a server only had to do one thing: get a file from the path from the server root specified by the URL path. HTML was just text, nothing more. Later, it became possible to add images. Twenty years later, a single "web page" can pull in 50 or more resources.

Along the way, servers began getting more complex and the Web turned from a static content medium into a dynamic content generation platform. First, there was the capability to perform simple queries, then to run scripts via a mechanism known as CGI. Finally, specialized protocols to run applications began to emerge.

In the Python world, things started moving very early on. Zope, the first Python application server, was open sourced in 1998, when opening up the code from a commercial project was still big news. Zope was ahead of its time in many ways and, though it has become less popular, it still survives and is in heavy use after all these years.

Zope was what we call a full-stack framework, which means it provides the whole range of services discussed in the previous section, and lots more. At the same time, other Python frameworks started coming out, but they all interoperated with web servers in different ways or used generic web server gateways that required some code to interface. That meant that the choice of framework would limit the choice of web servers and vice versa.

In 2003, a protocol named WSGI (Web Server Gateway Interface) was proposed to provide a standard interface between web servers and Python web applications. Inspired by the Java Servlet specification, the WSGI protocol took off after some time and is now the de facto way to connect Python applications with web servers.

A server implementing the WSGI specification can receive a request, pass it to the web application, and send the application's response back to the client. Applications can be stacked, so it's possible to have middleware that transforms the application response before it's returned.

Today, most Python web frameworks implement or support WSGI, and there are some libraries and toolkits that make it easier to create WSGI-compatible frameworks.

Installing Python Packages

When discussing some of the most notable frameworks back in Chapter 2, there was a Quick Start section for each. For people who don't have previous experience with Python packaging, we include some information about installing packages in this appendix.

For more information, consult the official Python packaging user guide, located at http://bit.ly/1O2NiWu.

Requirements for Installing Packages

To be able to install Python packages, you need to install a few requirements: *pip*, *setuptools*, and *wheel*.

Most likely, pip will already be installed on your system, but you will need to upgrade it before use. On Linux or OSX, upgrade with:

```
pip install -U pip setuptools
```

For Windows, the command should be:

```
python -m pip install -U pip setuptools
```

After that, add wheel using:

```
pip install wheel
```

If possible, use a clean Python installation, rather than one managed by your operating system's package manager. Updates in that case can become harder due to different OS development cycles.

Using pip

The recommended installer for Python is pip, and it is most frequently used to install packages from the Python Package Index. To install a package:

```
pip install 'package'
```

It's possible to install a specific version of a package, using:

```
pip install 'package==1.0'
```

It's also possible to install several packages using a *requirements file*, which is a text file where the required packages are listed, one package specifier per line (that is, a package name or a name and version expression):

```
pip install -r requirements.txt
```

Virtual Environments

A "virtual environment" for Python allows packages to be installed in an isolated location, thus preventing version conflicts and unwanted upgrades. For example, an already working application could require an earlier version of a popular library, but a new application requires the newer version. With a single Python installation, you risk affecting the application that is known to work fine when doing the upgrade.

Virtual environments avoid this kind of conflict, allowing you to have multiple "virtual" installations of the same Python version, each with its own libraries and installation directories.

The most popular tool for creating virtual environments is *virtualenv*, which supports all active Python versions and includes pip and setuptools by default on each virtual environment. Python 3.3 and newer includes *pyvenv*, which performs a similar function.

Either of those tools makes it easy to create a virtual environment. For virtualenv, *virtualenv <directory>* will set up a virtual environment, including Python binaries, inside the chosen directory. For pyvenv, *pyvenv <directory>* will do the same.

About the Author

Carlos de la Guardia has been doing web development with Python since 2000. He loves the language and the communities around it. He has contributed to Pyramid, SubstanceD, and other open source projects.