Explore Flask Documentation

Release 1.0

Robert Picard

February 26, 2017

Contents

Explore Flask is a book about best practices and patterns for developing web applications with Flask. The book was funded by 426 backers on Kickstarter in July 2013.

I finally released the book, after spending almost a year working on it. Almost immediately I was tired of managing distribution and limiting the book's audience by putting it behind a paywall. I didn't write a book to run a business, I wrote it to put some helpful content out there and help grow the Flask community.

In June of 2014, soon after finishing the book, I reformatted it for the web and released it here for free. No payment or donation or anything required. Just enjoy!

CHAPTER 1

About the author

My name is Robert Picard. I'm a security engineer at Addepar and a Flask enthusiast. I like Flask for its simplicity in the face of frameworks like Django that try and be everything to everyone. That model works for a lot of people, but not for me.

If you want to get in touch, feel free to send me an email at robert@robert.io or connect on twitter @__rlp. If you have feedback on the book, check out the GitHub repository too.

CHAPTER 2

Contents

Preface

This book is a collection of the best practices for using Flask. There are a lot of pieces to the average Flask application. You'll often need to interact with a database and authenticate users, for example. In the coming pages I'll do my best to explain the "right way" to do this sort of stuff. My recommendations aren't always going to apply, but I'm hoping that they'll be a good option most of the time.

Assumptions

In order to present you with more specific advice, I've written this book with a few fundamental assumptions. It's important to keep this in mind when you're reading and applying these recommendations to your own projects.

Audience

The content of this book builds upon the information in the official documentation. I highly recommend that you go through the user guide and follow along with the tutorial. This will give you a chance to become familiar with the vocabulary of Flask. You should understand what views are, the basics of Jinja templating and other fundamental concepts defined for beginners. I've tried to avoid overlap with the information already available in the user guide, so if you read this book first, there's a good chance that you'll find yourself lost (is that an oxymoron?).

With all of that said, the topics in this book aren't highly advanced. The goal is just to highlight best practices and patterns that will make development easier for you. While I'm trying to avoid too much overlap with the official documentation, you may find that I reiterate certain concepts to make sure that they're familiar. You shouldn't need to have the beginner's tutorial open while you read this.

Versions

Python 2 versus Python 3

As I write this, the Python community is in the midst of a transition from Python 2 to Python 3. The official stance of the Python Software Foundation is as follows:

> Python 2.x is the status quo, Python 3.x is the present and future of the language. [1]

As of version 0.10, Flask runs with Python 3.3. When I asked Armin Ronacher about whether new Flask apps should begin using Python 3, he said that he's not yet recommending it to people.

[1] Source: The Python wiki

> I'm not using it myself currently, and I don't ever recommend to people things that I don't believe in myself, so I'm very cautious about recommending Python 3.
>
> —Armin Ronacher, creator of Flask [2]

One reason for holding off on Python 3 is that many common dependencies haven't been ported yet. You don't want to build a project around Python 3 only to realize a few months down the line that you can't use packages X, Y and Z. It's possible that eventually Flask will officially recommend Python 3 for new projects, but for now it's all about Python 2.

Note: The Python 3 Wall of Superpowers tracks which major Python packages have been ported to Python 3.

Since this book is meant to provide practical advice, I think it makes sense to write with the assumption of Python 2. Specifically, I'll be writing the book with Python 2.7 in mind. Future updates may very well change this to evolve with the Flask community, but for now 2.7 is where we stand.

Flask version 0.10

At the time of writing this, 0.10 is the latest version of Flask (0.10.1 to be exact). Most of the lessons in this book aren't going to change with minor updates to Flask, but it's something to keep in mind nonetheless.

Living document

The content of this books is going to be updated on the fly, rather than with periodic releases. That is one of the benefits of putting the content out there for free, rather than putting it behind a walled garden. The web is a much more fluid distribution channel than print or even PDFs.

The book's source is hosted on GitHub and that is where "development" will be happening. Contributions and ideas are always welcome!

Conventions used in this book

Each chapter stands on its own

Each chapter in this book is an isolated lesson. Many books and tutorials are written as one long lesson. Generally this means that an example program or application is created and updated throughout the book to demonstrate concepts and lessons. Instead, examples are included in each lesson to demonstrate the concepts, but the examples from different chapters aren't meant to be combined into one large project.

Formatting

Footnotes will be used for citations so you don't think I'm making things up. [3]

Italic text will be used to denote a file name.

Bold text will be used to denote a new or important term.

Warning: Common pitfalls that could cause major problems will be shown in a warning box.

[2] Source: My conversation with Armin Ronacher

[3] See, it *must* be true!

Note: Supplemental information will appear in note boxes.

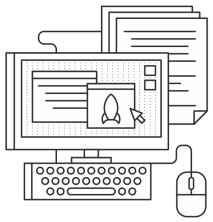
Easter eggs

Six backer names from the Kickstarter campaign have been encoded and sprinkled around the book. If you find all six and email the locations to me, I'll send you an extraordinarily mediocre prize. No hints.

Summary

- This book contains recommendations for using Flask.
- I'm assuming that you've gone through the Flask tutorial.
- I'm using Python 2.7.
- I'm using Flask 0.10.
- I'll do my best to keep the content of the book up-to-date.
- Each chapter in this book stands on its own.
- There are a few ways that I'll use formatting to convey additional information about the content.
- Summaries will appear as concise lists of takeaways from the chapters.

Coding conventions



There are a number of conventions in the Python community to guide the way you format your code. If you've been developing with Python for a while, then you might already be familiar with some of these conventions. I'll keep things brief and leave a few URLs where you can find more information if you haven't come across these topics before.

Let's have a PEP rally!

A **PEP** is a "Python Enhancement Proposal." These proposals are indexed and hosted at python.org. In the index, PEPs are grouped into a number of categories, including meta-PEPs, which are more informational than technical. The technical PEPs, on the other hand, often describe things like improvements to Python's internals.

There are a few PEPs, like PEP 8 and PEP 257 that are meant to guide the way we write our code. PEP 8 contains coding style guidelines. PEP 257 contains guidelines for docstrings, the generally accepted method of documenting code.

PEP 8: Style Guide for Python Code

PEP 8 is the official style guide for Python code. I recommend that you read it and apply its recommendations to your Flask projects (and all of your other Python code). Your code will be much more approachable when it starts growing to many files with hundreds, or thousands, of lines of code. The PEP 8 recommendations are all about having more readable code. Plus, if your project is going to be open source, potential contributors will likely expect and be comfortable working on code written with PEP 8 in mind.

One particularly important recommendation is to use 4 spaces per indentation level. No real tabs. If you break this convention, it'll be a burden on you and other developers when switching between projects. That sort of inconsistency is a pain in any language, but white-space is especially important in Python, so switching between real tabs and spaces could result in any number of errors that are a hassle to debug.

PEP 257: Docstring Conventions

PEP 257 covers another Python standard: **docstrings**. You can read the definition and recommendations in the PEP itself, but here's an example to give you an idea of what a docstring looks like:

```
def launch_rocket():
    """Main launch sequence director.

    Locks seatbelts, initiates radio and fires engines.
    """
    # [...]
```

These kinds of docstrings can be used by software such as Sphinx to generate documentation files in HTML, PDF and other formats. They also make it easier to understand your code.

Note:

- PEP 8
- PEP 257
- Sphinx, the documentation generator created by the same folks who brought us Flask

---

Relative imports

Relative imports make life a little easier when developing Flask apps. The premise is simple. Let's say you want to import the `User` model from the module *myapp/models.py*. You might think to use the app's package name, i.e. `myapp.models`. Using relative imports, you would indicate the location of the target module relative to the source. To do this we use a dot notation where the first dot indicates the current directory and each subsequent dot represents the next parent directory. Listing~ illustrates the diffence in syntax.

```
# myapp/views.py

# An absolute import gives us the User model
from myapp.models import User

# A relative import does the same thing
from .models import User
```

The advantage of this method is that the package becomes a heck of a lot more modular. Now you can rename your package and re-use modules from other projects without the need to update the hard-coded import statements.

In my research I came across a Tweet that illustrates the benefit of relative imports.

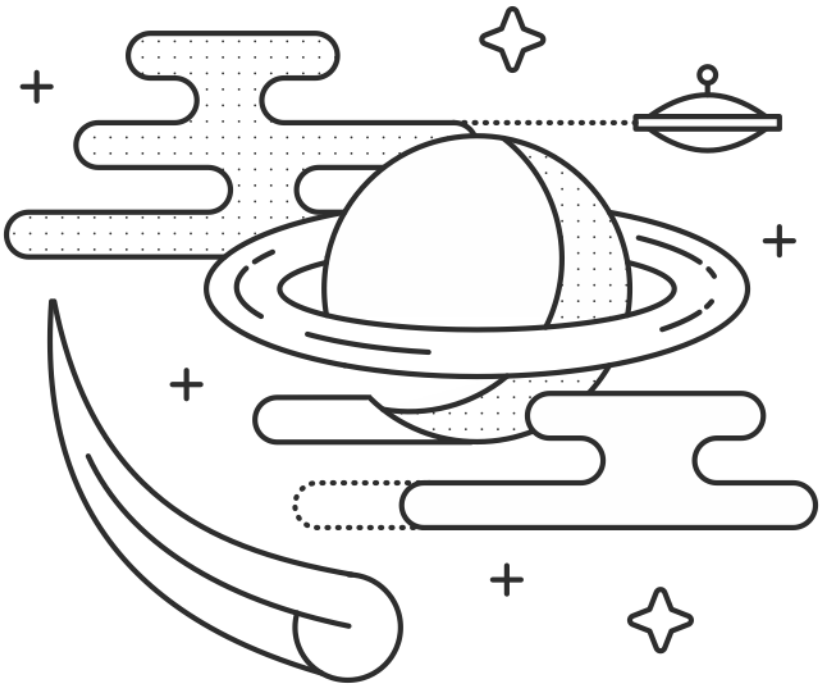
> Just had to rename our whole package. Took 1 second. Package relative imports FTW!
>
> —David Beazley, @dabeaz

Note: You can read a little more about the syntax for relative imports from this section in PEP 328.

Summary

- Try to follow the coding style conventions laid out in PEP 8.
- Try to document your app with docstrings as defined in PEP 257.
- Use relative imports to import your app's internal modules.

Environment



Your application is probably going to require a lot of software to function properly. If it doesn't at least require the Flask package, you may be reading the wrong book. Your application's **environment** is essentially all of the things that need to be around when it runs. Lucky for us, there are a number of things that we can do to make managing our environment much less complicated.

Use virtualenv to manage your environment

virtualenv is a tool for isolating your application in what is called a **virtual environment**. A virtual environment is a directory that contains the software on which your application depends. A virtual environment also changes your environment variables to keep your development environment contained. Instead of downloading packages, like Flask, to your system-wide — or user-wide — package directories, we can download them to an isolated directory used only for our current application. This makes it easy to specify which Python binary to use and which dependencies we want to have available on a per project basis.

Virtualenv also lets you use different versions of the same package for different projects. This flexibility may be important if you're working on an older system with several projects that have different version requirements.

When using virtualenv, you'll generally have only a few Python packages installed globally on your system. One of these will be virtualenv itself. You can install the `virtualenv` package with Pip.

Once you have virtualenv on your system, you can start creating virtual environments. Navigate to your project directory and run the `virtualenv` command. It takes one argument, which is the destination directory of the virtual environment. Listing~ shows what this looks like.

```
$ virtualenv venv
New python executable in venv/bin/python
Installing Setuptools...........[...].....done.
Installing Pip.................[...].....done.
$
```

virtualenv creates a new directory where the dependencies will be installed.

Once the new virtual environment has been created, you must activate it by sourcing the *bin/activate* script that was created inside the virtual environment.

```
$ which python
/usr/local/bin/python
$ source venv/bin/activate
(venv)$ which python
/Users/robert/Code/myapp/venv/bin/python
```

The *bin/activate* script makes some changes to your shell's environment variables so that everything points to the new virtual environment instead of your global system. You can see the effect in code block above. After activation, the `python` command refers to the Python binary inside the virtual environment. When a virtual environment is active, dependencies installed with Pip will be downloaded to that virtual environment instead of the global system.

You may notice that the shell prompt has been changed too. virtualenv prepends the name of the currently activated virtual environment, so you know that you're not working on the global system.

You can deactivate your virtual environment by running the `deactivate` command.

```
(venv)$ deactivate
$
```

virtualenvwrapper

virtualenvwrapper is a package used to manage the virtual environments created by virtualenv. I didn't want to mention this tool until you had seen the basics of virtualenv so that you understand what it's improving upon and understand why you should use it.

That virtual environment directory created in Listing~ref{code:venv_create} adds clutter to your project repository. You only interact with it directly when activating the virtual environment and it shouldn't be in version control, so there's no need to have it in there. The solution is to use virtualenvwrapper. This package keeps all of your virtual environments out of the way in a single directory, usually _~/.virtualenvs/_ by default.

To install virtualenvwrapper, follow the instructions in the documentation.

> **Warning:** Make sure that you've deactivated all virtual environments before installing virtualenvwrapper. You want it installed globally, not in a pre-existing environment.

Now, instead of running `virtualenv` to create an environment, you'll run `mkvirtualenv`:

```
$ mkvirtualenv rocket
New python executable in rocket/bin/python
Installing setuptools...........[...].....done.
Installing pip.................[...].....done.
(rocket)$
```

`mkvirtualenv` creates a directory in your virtual environments folder and activates it for you. Just like with plain old `virtualenv`, `python` and `pip` now point to that virtual environment instead of the system binaries. To activate a particular environment, use the command: `workon [environment name]`. `deactivate` still deactivates the environment.

Keeping track of dependencies

As a project grows, you'll find that the list of dependencies grows with it. It's not uncommon to need dozens of Python packages installed to run a Flask application. The easiest way to manage these is with a simple text file. Pip can generate a text file listing all installed packages. It can also read in this list to install each of them on a new system, or in a freshly minted environment.

pip freeze

requirements.txt is a text file used by many Flask applications to list all of the packages needed to run an application. This code block shows how to create this file and the following one shows how to use that text file to install your dependencies in a new environment.

```
(rocket)$ pip freeze > requirements.txt
```

```
$ workon fresh-env
(fresh-env)$ pip install -r requirements.txt
[...]
Successfully installed flask Werkzeug Jinja2 itsdangerous markupsafe
Cleaning up...
(fresh-env)$
```

Manually tracking dependencies

As your project grows, you may find that certain packages listed by `pip freeze` aren't actually needed to run the application. You'll have packages that are installed for development only. `pip freeze` doesn't discriminate between the two, it just lists the packages that are currently installed. As a result, you may want to manually track your dependencies as you add them. You can separate those packages needed to run your application and those needed to develop your application into *require_run.txt* and *require_dev.txt* respectively.

Version control

Pick a version control system and use it. I recommend Git. From what I've seen, Git is the most popular choice for new projects these days. Being able to delete code without worrying about making an irreversible mistake is invaluable.

You'll be able to keep your project free of those massive blocks of commented out code, because you can delete it now and revert that change later should the need arise. Plus, you'll have backup copies of your entire project on GitHub, Bitbucket or your own Gitolite server.

What to keep out of version control

I usually keep a file out of version control for one of two reasons. Either it's clutter, or it's a secret. Compiled *.pyc* files and virtual environments — if you're not using virtualenvwrapper for some reason — are examples of clutter. They don't need to be in version control because they can be recreated from the *.py* files and your *requirements.txt* files respectively.

API keys, application secret keys and database credentials are examples of secrets. They shouldn't be in version control because their exposure would be a massive breach of security.

Note: When making security related decisions, I always like to assume that my repository will become public at some point. This means keeping secrets out and never assuming that a security hole won't be found because, "Who's going to guess that they can do that?" This kind of assumption is known as security by obscurity and it's a bad policy to rely on.

When using Git, you can create a special file called *.gitignore* in your repository. In it, list wildcard patterns to match against filenames. Any filename that matches one of the patterns will be ignored by Git. I recommend using the *.gitignore* shown in Listing~ to get you started.

```
1 *.pyc
2 instance/
```

Instance folders are used to make secret configuration variables available to your application in a more secure way. We'll talk more about them later.

Note: You can read more about *.gitignore* here: http://git-scm.com/docs/gitignore

---

Debugging

Debug Mode

Flask comes with a handy feature called debug mode. To turn it on, you just have to set `debug = True` in your development configuration. When it's on, the server will reload on code changes and errors will come with a stack trace and an interactive console.

> **Warning:** Take care not to enable debug mode in production. The interactive console enables arbitrary code execution and would be a massive security vulnerability if it was left on in the live site.

Flask-DebugToolbar

Flask-DebugToolbar is another great tool for debugging problems with your application. In debug mode, it overlays a side-bar onto every page in your application. The side bar gives you information about SQL queries, logging, versions, templates, configuration and other fun stuff that makes it easier to track down problems.

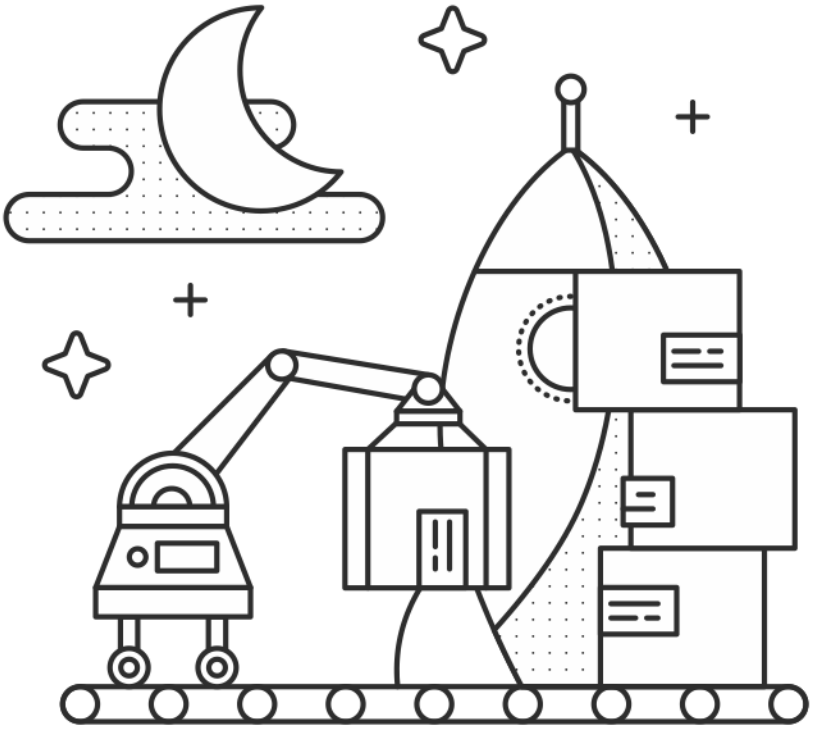
Note:

- Take a look at the quick start section on debug mode.
- There is some good information on handling errors, logging and working with other debuggers in the flask docs.

Summary

- Use virtualenv to keep your application's dependencies together.
- Use virtualenvwrapper to keep your virtual environments together.
- Keep track of dependencies with one or more text files.
- Use a version control system. I recommend Git.
- Use .gitignore to keep clutter and secrets out of version control.
- Debug mode can give you information about problems in development.
- The Flask-DebugToolbar extension will give you even more of that information.

Organizing your project



Flask leaves the organization of your application up to you. This is one of the reasons I liked Flask as a beginner, but it does mean that you have to put some thought into how to structure your code. You could put your entire application in one file, or have it spread across multiple packages. There are a few organizational patterns that you can follow to make development and deployment easier.

Definitions

Let's define some of the terms that we'll run into in this chapter.

Repository - This is the base folder where your applications sits. This term traditionally refers to version control systems, which you should be using. When I refer to your repository in this chapter, I'm talking about the root directory of your project. You probably won't need to leave this directory when working on your application.

Package - This refers to a Python package that contains your application's code. I'll talk more about setting up your app as a package in this chapter, but for now just know that the package is a sub-directory of the repository.

Module - A module is a single Python file that can be imported by other Python files. A package is essentially multiple modules packaged together.

Note:

- Read more about Python modules in Python tutorial.
- That same page has a section on packages.

Organization patterns

Single module

A lot of the Flask examples that you'll come across will keep all of the code in a single file, often *app.py*. This is great for quick projects (like the ones used for tutorials), where you just need to serve a few routes and you've got less than a few hundred lines of application code.

```
app.py
config.py
requirements.txt
static/
templates/
```

Application logic would sit in *app.py* for the example in Listing~.

Package

When you're working on a project that's a little more complex, a single module can get messy. You'll need to define classes for models and forms, and they'll get mixed in with the code for your routes and configuration. All of this can frustrate development. To solve this problem, we can factor out the different components of our app into a group of inter-connected modules — a package.

```
config.py
requirements.txt
run.py
instance/
    config.py
yourapp/
```

```
    __init__.py
    views.py
    models.py
    forms.py
    static/
    templates/
```

The structure shown in this listing allows you to group the different components of your application in a logical way. The class definitions for models are together in *models.py*, the route definitions are in *views.py* and forms are defined in *forms.py* (we have a whole chapter for forms later).

This table provides a basic rundown of the components you'll find in most Flask applications. You'll probably end up with a lot of other files in your repository, but these are common to most Flask applications.

run.py	This is the file that is invoked to start up a development server. It gets a copy of the app from your package and runs it. This won't be used in production, but it will see a lot of mileage in development.
requirements.txt	This file lists all of the Python packages that your app depends on. You may have separate files for production and development dependencies.
config.py	This file contains most of the configuration variables that your app needs.
/instance/config.py	This file contains configuration variables that shouldn't be in version control. This includes things like API keys and database URIs containing passwords. This also contains variables that are specific to this particular instance of your application. For example, you might have `DEBUG = False` in config.py, but set `DEBUG = True` in instance/config.py on your local machine for development. Since this file will be read in after config.py, it will override it and set `DEBUG = True`.
/yourapp/	This is the package that contains your application.
/yourapp/__init__.py	This file initializes your application and brings together all of the various components.
/yourapp/views.py	This is where the routes are defined. It may be split into a package of its own (*yourapp/views/*) with related views grouped together into modules.
/yourapp/models.py	This is where you define the models of your application. This may be split into several modules in the same way as views.py.
/yourapp/static/	This directory contains the public CSS, JavaScript, images and other files that you want to make public via your app. It is accessible from yourapp.com/static/ by default.
/yourapp/templates/	This is where you'll put the Jinja2 templates for your app.

Blueprints

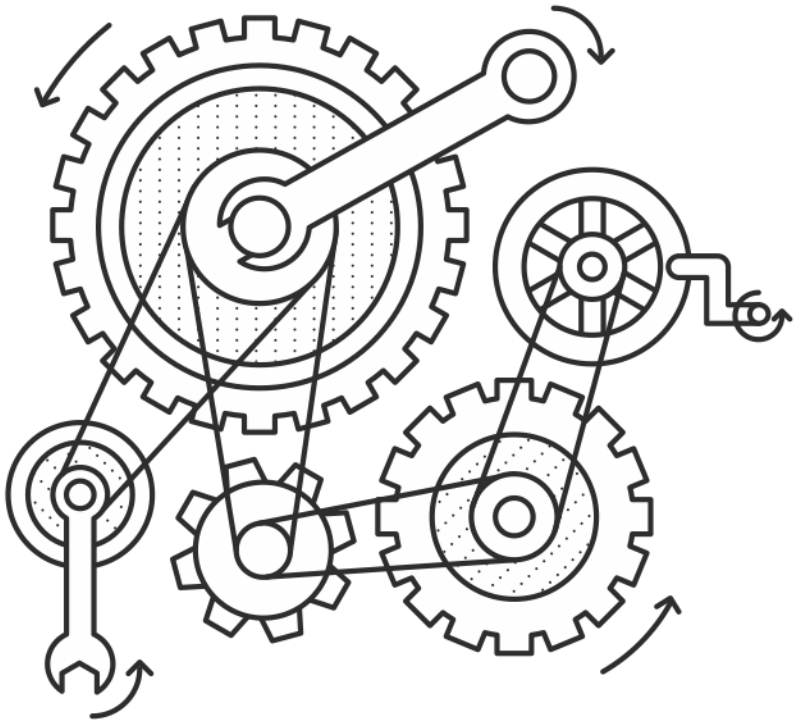
At some point you may find that you have a lot of related routes. If you're like me, your first thought will be to split *views.py* into a package and group those views into modules. When you're at this point, it may be time to factor your application into blueprints.

Blueprints are essentially components of your app defined in a somewhat self-contained manner. They act as apps within your application. You might have different blueprints for the admin panel, the front-end and the user dashboard. This lets you group views, static files and templates by components, while letting you share models, forms and other aspects of your application between these components. We'll talk about using Blueprints to organize your application soon.

Summary

- Using a single module for your application is good for quick projects.
- Using a package for your application is good for projects with views, models, forms and other components.
- Blueprints are a great way to organize projects with several distinct components.

Configuration



When you're learning Flask, configuration seems simple. You just define some variables in *config.py* and everything works. That simplicity starts to fade away when you have to manage configuration for a production application. You may need to protect secret API keys or use different configurations for different environments (e.g. development and production environments). In this chapter we'll go over some advanced Flask features that makes this managing configuration easier.

The simple case

A simple application may not need any of these complicated features. You may just need to put *config.py* in the root of your repository and load it in *app.py* or *yourapp/__init__.py*

The *config.py* file should contain one variable assignment per line. When your app is initialized, the variables in *config.py* are used to configure Flask and its extensions are accessible via the `app.config` dictionary – e.g. `app.config["DEBUG"]`.

```
DEBUG = True # Turns on debugging features in Flask
BCRYPT_LEVEL = 12 # Configuration for the Flask-Bcrypt extension
MAIL_FROM_EMAIL = "robert@example.com" # For use in application emails
```

Configuration variables can be used by Flask, extensions or you. In this example, we could use `app.config["MAIL_FROM_EMAIL"]` whenever we needed the default "from" address for a transactional email

– e.g. password resets. Putting that information in a configuration variable makes it easy to change it in the future.

```
# app.py or app/__init__.py
from flask import Flask

app = Flask(__name__)
app.config.from_object('config')

# Now we can access the configuration variables via app.config["VAR_NAME"].
```

Variable	Decription	Recommendation
DEBUG	Gives you some handy tools for debugging errors. This includes a web-based stack trace and interactive Python console for errors.	Should be set to `True` in development and `False` in production.
SECRET_KEY	This is a secret key that is used by Flask to sign cookies. It's also used by extensions like Flask-Bcrypt. You should define this in your instance folder to keep it out of version control. You can read more about instance folders in the next section.	This should be a complex random value.
BCRYPT_LEVEL	If you're using Flask-Bcrypt to hash user passwords, you'll need to specify the number of "rounds" that the algorithm executes in hashing a password. If you aren't using Flask-Bcrypt, you should probably start. The more rounds used to hash a password, the longer it'll take for an attacker to guess a password given the hash. The number of rounds should increase over time as computing power increases.	Later in this book we'll cover some of the best practices for using Bcrypt in your Flask application.

Warning: Make sure `DEBUG` is set to `False` in production. Leaving it on will allow users to run arbitrary Python code on your server.

Instance folder

Sometimes you'll need to define configuration variables that contain sensitive information. We'll want to separate these variables from those in *config.py* and keep them out of the repository. You may be hiding secrets like database passwords and API keys, or defining variables specific to a given machine. To make this easy, Flask gives us a feature called **instance folders**. The instance folder is a sub-directory of the repository root and contains a configuration file specifically for this instance of the application. We don't want to commit it into version control.

```
config.py
requirements.txt
run.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  templates/
  static/
```

Using instance folders

To load configuration variables from an instance folder, we use `app.config.from_pyfile()`. If we set `instance_relative_config=True` when we create our app with the `Flask()` call,

`app.config.from_pyfile()` will load the specified file from the *instance/* directory.

```
# app.py or app/__init__.py

app = Flask(__name__, instance_relative_config=True)
app.config.from_object('config')
app.config.from_pyfile('config.py')
```

Now, we can define variables in *instance/config.py* just like you did in *config.py*. You should also add the instance folder to your version control system's ignore list. To do this with Git, you would add `instance/` on a new line in *.gitignore*.

Secret keys

The private nature of the instance folder makes it a great candidate for defining keys that you don't want exposed in version control. These may include your app's secret key or third-party API keys. This is especially important if your application is open source, or might be at some point in the future. We usually want other users and contributors to use their own keys.

```
# instance/config.py

SECRET_KEY = 'Sm9obiBTY2hyb20ga2lja3MgYXNz'
STRIPE_API_KEY = 'SmFjb2IgS2FwbGFuLU1vc3MgaXMgYSBoZXJv'
SQLALCHEMY_DATABASE_URI= \
"postgresql://user:TWljaGHFgiBCYXJ0b3N6a2lld2ljeiEh@localhost/databasename"
```

Minor environment-based configuration

If the difference between your production and development environments are pretty minor, you may want to use your instance folder to handle the configuration changes. Variables defined in the *instance/config.py* file can override the value in *config.py*. You just need to make the call to `app.config.from_pyfile()` after `app.config.from_object()`. One way to take advantage of this is to change the way your app is configured on different machines.

```
# config.py

DEBUG = False
SQLALCHEMY_ECHO = False


# instance/config.py
DEBUG = True
SQLALCHEMY_ECHO = True
```

In production, we would leave the variables in the above listing out of *instance/config.py* and it would fall back to the values defined in *config.py*.

Note:

- Read more about Flask-SQLAlchemy's configuration keys

Configuring based on environment variables

The instance folder shouldn't be in version control. This means that you won't be able to track changes to your instance configurations. That might not be a problem with one or two variables, but if you have finely tuned configurations for various environments (production, staging, development, etc.) you don't want to risk losing that.

Flask gives us the ability to choose a configuration file on load based on the value of an environment variable. This means that we can have several configuration files in our repository and always load the right one. Once we have several configuration files, we can move them to their own `config` directory.

```
requirements.txt
run.py
config/
  __init__.py # Empty, just here to tell Python that it's a package.
  default.py
  production.py
  development.py
  staging.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  static/
  templates/
```

In this listing we have a few different configuration files.

config/default.py	Default values, to be used for all environments or overridden by individual environments. An example might be setting DEBUG = False in config/default.py and DEBUG = True in config/development.py.
config/development.py	Values to be used during development. Here you might specify the URI of a database sitting on localhost.
config/production.py	Values to be used in production. Here you might specify the URI for your database server, as opposed to the localhost database URI used for development.
config/staging.py	Depending on your deployment process, you may have a staging step where you test changes to your application on a server that simulates a production environment. You'll probably use a different database, and you may want to alter other configuration values for staging applications.

To decide which configuration file to load, we'll call `app.config.from_envvar()`.

```
# yourapp/__init__.py

app = Flask(__name__, instance_relative_config=True)

# Load the default configuration
app.config.from_object('config.default')

# Load the configuration from the instance folder
app.config.from_pyfile('config.py')

# Load the file specified by the APP_CONFIG_FILE environment variable
# Variables defined here will override those in the default configuration
app.config.from_envvar('APP_CONFIG_FILE')
```

The value of the environment variable should be the absolute path to a configuration file.

How we set this environment variable depends on the platform in which we're running the app. If we're running on a

regular Linux server, we can set up a shell script that sets our environment variables and runs *run.py*.

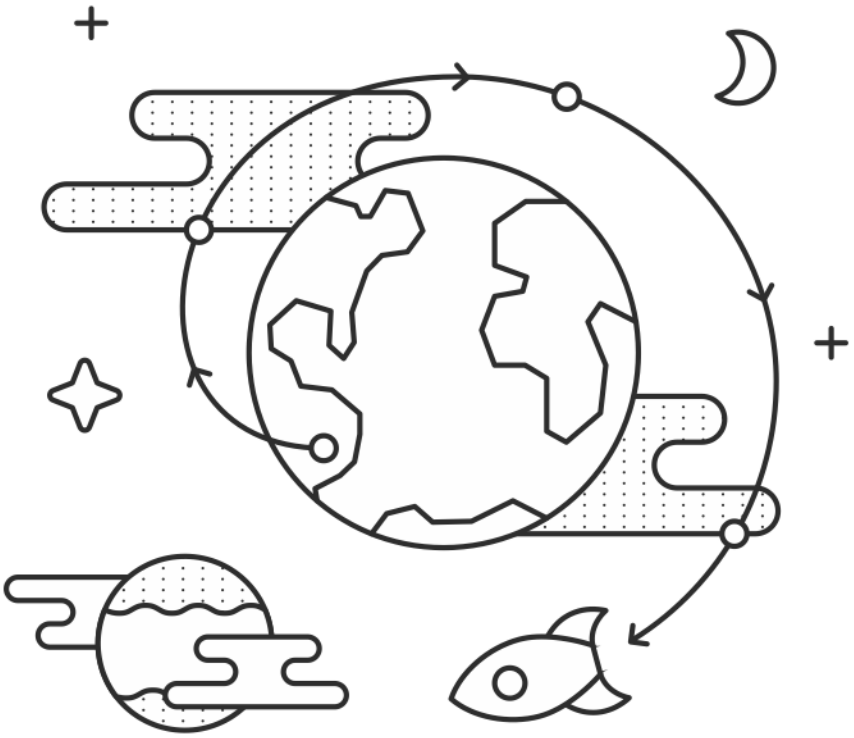
```
# start.sh

export APP_CONFIG_FILE=/var/www/yourapp/config/production.py
python run.py
```

start.sh is unique to each environment, so it should be left out of version control. On Heroku, we'll want to set the environment variables with the Heroku tools. The same idea applies to other PaaS platforms.

Summary

- A simple app may only need one configuration file: *config.py*.
- Instance folders can help us hide secret configuration values.
- Instance folders can be used to alter an application's configuration for a specific environment.
- We should use environment variables and `app.config.from_envvar()` for more complicated environment-based configurations.

Advanced patterns for views and routing



View decorators

Python decorators are functions that are used to transform other functions. When a decorated function is called, the decorator is called instead. The decorator can then take action, modify the arguments, halt execution or call the original function. We can use decorators to wrap views with code we'd like to run before they are executed.

```
@decorator_function
def decorated():
    pass
```

If you've gone through the Flask tutorial, the syntax in this code block might look familiar to you. `@app.route` is a decorator used to match URLs to view functions in Flask apps.

Let's take a look at some other decorators you can use in your Flask apps.

Authentication

The Flask-Login extension makes it easy to implement a login system. In addition to handling the details of user authentication, Flask-Login gives us a decorator to restrict certain views to authenticated users: `@login_required`.

```
# app.py

from flask import render_template
from flask_login import login_required, current_user


@app.route('/')
def index():
    return render_template("index.html")

@app.route('/dashboard')
@login_required
def account():
    return render_template("account.html")
```

Warning: `@app.route` should always be the outermost view decorator.

Only an authenticated user will be able to access the */dashboard* route. We can configure Flask-Login to redirect unauthenticated users to a login page, return an HTTP 401 status or anything else we'd like it to do with them.

Note: Read more about using Flask-Login in the official docs.

Caching

Imagine that an article mentioning our application just appeared on CNN and some other news sites. We're getting thousands of requests per second. Our homepage makes several trips to the database for each request, so all of this attention is slowing things down to a crawl. How can we speed things up quickly, so all of these visitors don't miss out on our site?

There are a lot of good answers, but this section is about caching, so we'll talk about that. Specifically, we're going to use the Flask-Cache extension. This extension provides us with a decorator that we can use on our index view to cache the response for some period of time.

Flask-Cache can be configured to work with a bunch of different caching backends. A popular choice is Redis, which is easy to set-up and use. Assuming Flask-Cache is already configured, this code block shows what our decorated view would look like.

```
# app.py

from flask_cache import Cache
from flask import Flask

app = Flask()

# We'd normally include configuration settings in this call
cache = Cache(app)

@app.route('/')
@cache.cached(timeout=60)
def index():
    [...] # Make a few database calls to get the information we need
    return render_template(
        'index.html',
        latest_posts=latest_posts,
        recent_users=recent_users,
        recent_photos=recent_photos
    )
```

Now the function will only be run once every 60 seconds, when the cache expires. The response will be saved in our cache and pulled from there for any intervening requests.

Note: Flask-Cache also lets us **memoize** functions — or cache the result of a function being called with certain arguments. You can even cache computationally expensive Jinja2 template snippets.

Custom decorators

For this section, let's imagine we have an application that charges users each month. If a user's account is expired, we'll redirect them to the billing page and tell them to upgrade.

```
# myapp/util.py

from functools import wraps
from datetime import datetime

from flask import flash, redirect, url_for

from flask_login import current_user

def check_expired(func):
    @wraps(func)
    def decorated_function(*args, **kwargs):
        if datetime.utcnow() > current_user.account_expires:
            flash("Your account has expired. Update your billing info.")
            return redirect(url_for('account_billing'))
        return func(*args, **kwargs)

    return decorated_function
```

10	When a function is decorated with `@check_expired`, `check_expired()` is called and the decorated function is passed as a parameter.
11	`@wraps` is a decorator that does some bookkeeping so that `decorated_function()` appears as `func()` for the purposes of documentation and debugging. This makes the behavior of the functions a little more natural.
12	`decorated_function` will get all of the args and kwargs that were passed to the original view function `func()`. This is where we check if the user's account is expired. If it is, we'll flash a message and redirect them to the billing page.
16	Now that we've done what we wanted to do, we run the decorated view function `func()` with its original arguments.

When we stack decorators, the topmost decorator will run first, then call the next function in line: either the view function or the next decorator. The decorator syntax is just a little syntactic sugar.

```
# This code:
@foo
@bar
def one():
    pass

r1 = one()

# is the same as this code:
def two():
    pass

two = foo(bar(two))
r2 = two()

r1 == r2 # True
```

This code block shows an example using our custom decorator and the `@login_required` decorator from the Flask-Login extension. We can use multiple decorators by stacking them.

```
# myapp/views.py

from flask import render_template

from flask_login import login_required

from . import app
from .util import check_expired

@app.route('/use_app')
@login_required
@check_expired
def use_app():
    """Use our amazing app."""
    # [...]
    return render_template('use_app.html')

@app.route('/account/billing')
@login_required
def account_billing():
    """Update your billing info."""
    # [...]
    return render_template('account/billing.html')
```

Now when a user tries to access */use_app*, `check_expired()` will make sure that their account hasn't expired

before running the view function.

Note: Read more about what the `wraps()` function does in the Python docs.

URL Converters

Built-in converters

When you define a route in Flask, you can specify parts of it that will be converted into Python variables and passed to the view function.

```
@app.route('/user/<username>')
def profile(username):
    pass
```

Whatever is in the part of the URL labeled `<username>` will get passed to the view as the username argument. You can also specify a converter to filter the variable before it's passed to the view.

```
@app.route('/user/id/<int:user_id>')
def profile(user_id):
    pass
```

In this code block, the URL *http://myapp.com/user/id/Q29kZUxlc3NvbiEh* will return a 404 status code – not found. This is because the part of the URL that is supposed to be an integer is actually a string.

We could have a second view that looks for a string as well. That would be called for */user/id/Q29kZUxlc3NvbiEh/* while the first would be called for */user/id/124*.

This table shows Flask's built-in URL converters.

string	Accepts any text without a slash (the default).
int	Accepts integers.
float	Like int but for floating point values.
path	Like string but accepts slashes.

Custom converters

We can also make custom converters to suit our needs. On Reddit — a popular link sharing site — users create and moderate communities for theme-based discussion and link sharing. Some examples are /r/python and /r/flask, denoted by the path in the URL: *reddit.com/r/python* and *reddit.com/r/flask* respectively. An interesting feature of Reddit is that you can view the posts from multiple subreddits as one by seperating the names with a plus-sign in the URL, e.g. *reddit.com/r/python+flask*.

We can use a custom converter to implement this feature in our own Flask apps. We'll take an arbitrary number of elements separated by plus-signs, convert them to a list with a `ListConverter` class and pass the list of elements to the view function.

```
# myapp/util.py

from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):

    def to_python(self, value):
        return value.split('+')
```

```
    def to_url(self, values):
        return '+'.join(BaseConverter.to_url(value)
                        for value in values)
```

We need to define two methods: `to_python()` and `to_url()`. As the names suggest, `to_python()` is used to convert the path in the URL to a Python object that will be passed to the view and `to_url()` is used by `url_for()` to convert arguments to their appropriate forms in the URL.

To use our `ListConverter`, we first have to tell Flask that it exists.

```
# /myapp/__init__.py

from flask import Flask

app = Flask(__name__)

from .util import ListConverter

app.url_map.converters['list'] = ListConverter
```

Warning: This is another chance to run into some circular import problems if your `util` module has a `from . import app` line. That's why I waited until app had been initialized to import `ListConverter`.
Now we can use our converter just like one of the built-ins. We specified the key in the dictionary as "list" so that's how we use it in `@app.route()`.

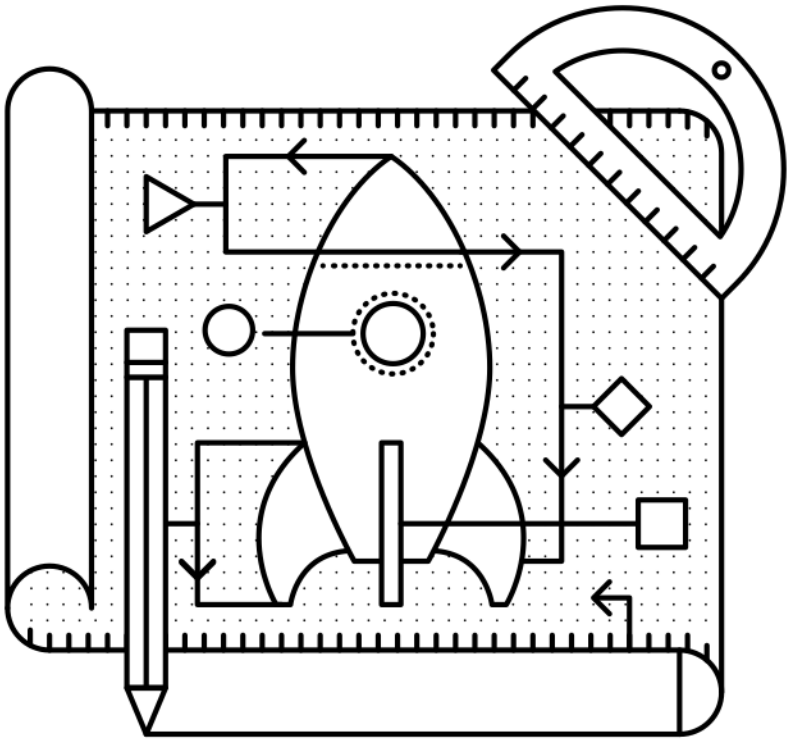
```
# myapp/views.py

from . import app

@app.route('/r/<list:subreddits>')
def subreddit_home(subreddits):
    """Show all of the posts for the given subreddits."""
    posts = []
    for subreddit in subreddits:
        posts.extend(subreddit.posts)

    return render_template('/r/index.html', posts=posts)
```

This should work just like Reddit's multi-reddit system. This same method can be used to make any URL converter we can dream of.

Summary

- The `@login_required` decorator from Flask-Login helps you limit views to authenticated users.
- The Flask-Cache extension gives you a bunch of decorators to implement various methods of caching.
- We can develop custom view decorators to help us organize our code and stick to DRY (Don't Repeat Yourself) coding principles.
- Custom URL converters can be a great way to implement creative features involving URL's.

Blueprints



What is a blueprint?

A blueprint defines a collection of views, templates, static files and other elements that can be applied to an application. For example, let's imagine that we have a blueprint for an admin panel. This blueprint would define the views for routes like */admin/login* and */admin/dashboard*. It may also include the templates and static files that will be served on those routes. We can then use this blueprint to add an admin panel to our app, be it a social network for astronauts or a CRM for rocket salesmen.

Why would you use blueprints?

The killer use-case for blueprints is to organize our application into distinct components. For a Twitter-like microblog, we might have a blueprint for the website pages, e.g. *index.html* and *about.html*. Then we could have another for the logged-in dashboard where we show all of the latest posts and yet another for our administrator's panel. Each distinct area of the site can be separated into distinct areas of the code as well. This lets us structure our app as several smaller "apps" that each do one thing.

Note: Read more about the benefits of using blueprints in "Why Blueprints" from the Flask docs.

Where do you put them?

Like everything with Flask, there are many ways that we can organize our app using blueprints. With blueprints, we can think of the choice as functional versus divisional (terms I'm borrowing from the business world).

Functional structure

With a functional structure, you organize the pieces of your app by what they do. Templates are grouped together in one directory, static files in another and views in a third.

```
yourapp/
    __init__.py
    static/
    templates/
        home/
        control_panel/
        admin/
    views/
        __init__.py
        home.py
        control_panel.py
        admin.py
    models.py
```

With the exception of *yourapp/views/__init__.py*, each of the *.py* files in the *yourapp/views/* directory from this listing is a blueprint. In *yourapp/__init__.py* we would import those blueprints and **register** them on our `Flask()` object. We'll look a little more at how this is implemented later in this chapter.

Note: At the time of writing this, the Flask website at http://flask.pocoo.org uses this structure. Take a look for yourself on GitHub.

Divisional

With the divisional structure, you organize the pieces of the application based on which part of the app they contribute to. All of the templates, views and static files for the admin panel go in one directory, and those for the user control panel go in another.

```
yourapp/
    __init__.py
    admin/
        __init__.py
        views.py
        static/
        templates/
    home/
        __init__.py
        views.py
        static/
        templates/
    control_panel/
        __init__.py
        views.py
        static/
```

```
        templates/
    models.py
```

With a divisional structure like the app in this listing, each directory under *yourapp/* is a separate blueprint. All of the blueprints are applied to the `Flask()` app in the top-level *__init__.py*

Which one is best?

The organizational structure you choose is largely a personal decision. The only difference is the way the hierarchy is represented – i.e. you can architect Flask apps with either methodology – so you should choose the one that makes sense to you.

If your app has largely independent pieces that only share things like models and configuration, divisional might be the way to go. An example might be a SaaS app that lets user's build websites. You could have blueprints in "divisions" for the home page, the control panel, the user's website, and the admin panel. These components may very well have completely different static files and layouts. If you're considering spinning off your blueprints as extensions or using them for other projects, a divisional structure will be easier to work with.

On the other hand, if the components of your app flow together a little more, it might be better represented with a functional structure. An example of this would be Facebook. If Facebook used Flask, it might have blueprints for the static pages (i.e. signed-out home, register, about, etc.), the dashboard (i.e. the news feed), profiles (*/robert/about* and */robert/photos*), settings (*/settings/security* and */settings/privacy*) and many more. These components all share a general layout and styles, but each has its own layout as well. The following listing shows a heavily abridged version of what Facebook might look like it if were built with Flask.

```
facebook/
    __init__.py
    templates/
        layout.html
        home/
            layout.html
            index.html
            about.html
            signup.html
            login.html
        dashboard/
            layout.html
            news_feed.html
            welcome.html
            find_friends.html
        profile/
            layout.html
            timeline.html
            about.html
            photos.html
            friends.html
            edit.html
        settings/
            layout.html
            privacy.html
            security.html
            general.html
    views/
        __init__.py
        home.py
        dashboard.py
        profile.py
```

```
        settings.py
    static/
        style.css
        logo.png
    models.py
```

The blueprints in *facebook/views/* are little more than collections of views rather than wholly independent components. The same static files will be used for the views in most of the blueprints. Most of the templates will extend a master template. A functional structure is a good way to organize this project.

How do you use them?

Basic usage

Let's take a look at the code for one of the blueprints from that Facebook example.

```
# facebook/views/profile.py

from flask import Blueprint, render_template

profile = Blueprint('profile', __name__)

@profile.route('/<user_url_slug>')
def timeline(user_url_slug):
    # Do some stuff
    return render_template('profile/timeline.html')

@profile.route('/<user_url_slug>/photos')
def photos(user_url_slug):
    # Do some stuff
    return render_template('profile/photos.html')

@profile.route('/<user_url_slug>/about')
def about(user_url_slug):
    # Do some stuff
    return render_template('profile/about.html')
```

To create a blueprint object, we import the `Blueprint()` class and initialize it with the arguments `name` and `import_name`. Usually `import_name` will just be `__name__`, which is a special Python variable containing the name of the current module.

We're using a functional structure for this Facebook example. If we were using a divisional structure, we'd want to tell Flask that the blueprint has its own template and static directories. This code block shows what that would look like.

```
profile = Blueprint('profile', __name__,
                    template_folder='templates',
                    static_folder='static')
```

We have now defined our blueprint. It's time to register it on our Flask app.

```
# facebook/__init__.py

from flask import Flask
from .views.profile import profile

app = Flask(__name__)
app.register_blueprint(profile)
```

Now the routes defined in *facebook/views/profile.py* (e.g. `/<user_url_slug>`) are registered on the application and act just as if you'd defined them with `@app.route()`.

Using a dynamic URL prefix

Continuing with the Facebook example, notice how all of the profile routes start with the `<user_url_slug>` portion and pass that value to the view. We want users to be able to access a profile by going to a URL like *https://facebook.com/john.doe*. We can stop repeating ourselves by defining a dynamic prefix for all of the blueprint's routes.

Blueprints let us define both static and dynamic prefixes. We can tell Flask that all of the routes in a blueprint should be prefixed with */profile* for example; that would be a static prefix. In the case of the Facebook example, the prefix is going to change based on which profile the user is viewing. Whatever text they choose is the URL slug of the profile which we should display; this is a dynamic prefix.

We have a choice to make when defining our prefix. We can define the prefix in one of two places: when we instantiate the `Blueprint()` class or when we register it with `app.register_blueprint()`.

```
# facebook/views/profile.py

from flask import Blueprint, render_template

profile = Blueprint('profile', __name__, url_prefix='/<user_url_slug>')

# [...]
```

```
# facebook/__init__.py

from flask import Flask
from .views.profile import profile

app = Flask(__name__)
app.register_blueprint(profile, url_prefix='/<user_url_slug>')
```

While there aren't any technical limitations to either method, it's nice to have the prefixes available in the same file as the registrations. This makes it easier to move things around from the top-level. For this reason, I recommend setting `url_prefix` on registration.

We can use converters to make the prefix dynamic, just like in `route()` calls. This includes any custom converters that we've defined. When using converters, we can pre-process the value given before handing it off to the view. In this case we'll want to grab the user object based on the URL slug passed into our profile blueprint. We'll do that by decorating a function with `url_value_preprocessor()`.

```
# facebook/views/profile.py

from flask import Blueprint, render_template, g

from ..models import User

# The prefix is defined on registration in facebook/__init__.py.
profile = Blueprint('profile', __name__)

@profile.url_value_preprocessor
def get_profile_owner(endpoint, values):
    query = User.query.filter_by(url_slug=values.pop('user_url_slug'))
    g.profile_owner = query.first_or_404()

@profile.route('/')
def timeline():
```

```
    return render_template('profile/timeline.html')

@profile.route('/photos')
def photos():
    return render_template('profile/photos.html')

@profile.route('/about')
def about():
    return render_template('profile/about.html')
```

We're using the `g` object to store the profile owner and `g` is available in the Jinja2 template context. This means that for a barebones case all we have to do in the view is render the template. The information we need will be available in the template.

```
{# facebook/templates/profile/photos.html #}

{% extends "profile/layout.html" %}

{% for photo in g.profile_owner.photos.all() %}
    <img src="{{ photo.source_url }}" alt="{{ photo.alt_text }}" />
{% endfor %}
```

Note:

- The Flask documentation has a great tutorial on using prefixes for internationalizing your URLs.

Using a dynamic subdomain

Many SaaS (Software as a Service) applications these days provide users with a subdomain from which to access their software. Harvest, for example, is a time tracking application for consultants that gives you access to your dashboard from yourname.harvestapp.com. Here I'll show you how to get Flask to work with automatically generated subdomains like this.

For this section I'm going to use the example of an application that lets users create their own websites. Imagine that our app has three blueprints for distinct sections: the home page where users sign-up, the user administration panel where the user builds their website and the user's website. Since these three parts are relatively unconnected, we'll organize them in a divisional structure.

```
sitemaker/
    __init__.py
    home/
        __init__.py
        views.py
        templates/
            home/
        static/
            home/
    dash/
        __init__.py
        views.py
        templates/
            dash/
        static/
            dash/
    site/
        __init__.py
```

```
        views.py
        templates/
            site/
        static/
            site/
    models.py
```

This table explains the different blueprints in this app.

URL	Route	Description
sitemaker.com	*sitemaker/home*	Just a vanilla blueprint. Views, templates and static files for *index.html*, *about.html* and *pricing.html*.
big-daddy.sitemaker.com	*sitemaker/site*	This blueprint uses a dynamic subdomain and includes the elements of the user's website. We'll go over some of the code used to implement this blueprint below.
big-daddy.sitemaker.com/admin	*sitemaker/admin*	This blueprint could use both a dynamic subdomain and a URL prefix by combining the techniques in this section with those from the previous section.

We can define our dynamic subdomain the same way we defined our URL prefix. Both options (in the blueprint directory or in the top-level *__init__.py*) are available, but once again we'll keep the definitions in *sitemaker/__init.py__*.

```
# sitemaker/__init__.py

from flask import Flask
from .site import site

app = Flask(__name__)
app.register_blueprint(site, subdomain='<site_subdomain>')
```

Since we're using a divisional structure, we'll define the blueprint in *sitemaker/site/__init__.py*.

```
# sitemaker/site/__init__py

from flask import Blueprint

from ..models import Site

# Note that the capitalized Site and the lowercase site
# are two completely separate variables. Site is a model
# and site is a blueprint.

site = Blueprint('site', __name__)

@site.url_value_preprocessor
def get_site(endpoint, values):
    query = Site.query.filter_by(subdomain=values.pop('site_subdomain'))
    g.site = query.first_or_404()

# Import the views after site has been defined. The views
# module will need to import 'site' so we need to make
# sure that we import views after site has been defined.
from . import views
```

Now we have the site information from the database that we'll use to display the user's site to the visitor who requests their subdomain.

To get Flask to work with subdomains, we'll need to specify the `SERVER_NAME` configuration variable.

```
# config.py

```

```
SERVER_NAME = 'sitemaker.com'
```

Note: A few minutes ago, as I was drafting this section, somebody in IRC said that their subdomains were working fine in development, but not in production. I asked if they had the `SERVER_NAME` configured, and it turned out that they had it in development but not production. Setting it in production solved their problem.

See the conversation between myself (imrobert in the log) and aplavin: http://dev.pocoo.org/irclogs/%23pocoo.2013-07-30.log

It was enough of a coincidence that I felt it warranted inclusion in the section.

Note: You can set both a subdomain and url_prefix. Think about how we would configure the blueprint in *sitemaker/dash* with the URL structure from the table above.

Refactoring small apps to use blueprints

I'd like to go over a brief example of the steps we can take to convert an app to use blueprints. We'll start off with a typical Flask app and restructure it.

```
config.txt
requirements.txt
run.py
U2FtIEJsYWNr/
  __init__.py
  views.py
  models.py
  templates/
  static/
tests/
```

The *views.py* file has grown to 10,000 lines of code! We've been putting off refactoring it, but it's finally time. The file contains the views for every section of our site. The sections are the home page, the user dashboard, the admin dashboard, the API and the company blog.

Step 1: Divisional or functional?

This application is made up of very distinct sections. Templates and static files probably aren't going to be shared between the user dashboard and the company blog, for example. We'll go with a divisional structure.

Step 2: Move some files around

Warning: Before you make any changes to your app, commit everything to version control. You don't want to accidentally delete something for good.

Next we'll go ahead and create the directory tree for our new app. We can start by creating a folder for each blueprint within the package directory. Then we'll copy *views.py*, *static/* and *templates/* in their entirety to each blueprint directory. We can then remove them from the top-level package directory.

```
config.txt
requirements.txt
run.py
U2FtIEJsYWNr/
  __init__.py
  home/
    views.py
    static/
    templates/
  dash/
    views.py
    static/
    templates/
  admin/
    views.py
    static/
    templates/
  api/
    views.py
    static/
    templates/
  blog/
    views.py
    static/
    templates/
  models.py
tests/
```

Step 3: Cut the crap

Now we can go into each blueprint and remove the views, static files and templates that don't apply to that blueprint. How you go about this step largely depends on how your app was organized to begin with.

The end result should be that each blueprint has a *views.py* file with all of the views for that blueprint. No two blueprints should define a view for the same route. Each *templates/* directory should only include the templates for the views in that blueprint. Each *static/* directory should only include the static files that should be exposed by that blueprint.

Note: Make it a point to eliminate all unnecessary imports. It's easy to forget about them, but at best they clutter your code and at worst they slow down your application.

Step 4: Blueprint...ifi...cation or something

This is the part where we turn our directories into blueprints. The key is in the *__init__.py* files. For starters, let's take a look at the definition of the API blueprint.

```
# U2FtIEJsYWNr/api/__init__.py

from flask import Blueprint

api = Blueprint(
    'site',
    __name__,
    template_folder='templates',
```

```
    static_folder='static'
)

from . import views
```

Next we can register this blueprint in the U2FtIEJsYWNr package's top-level *__init__.py* file.

```
# U2FtIEJsYWNr/__init__.py

from flask import Flask
from .api import api

app = Flask(__name__)

# Puts the API blueprint on api.U2FtIEJsYWNr.com.
app.register_blueprint(api, subdomain='api')
```

Make sure that the routes are registered on the blueprint now rather than the app object.

```
# U2FtIEJsYWNr/views.py

from . import app

@app.route('/search', subdomain='api')
def api_search():
    pass
```

```
# U2FtIEJsYWNr/api/views.py

from . import api

@api.route('/search')
def search():
    pass
```

Step 5: Enjoy

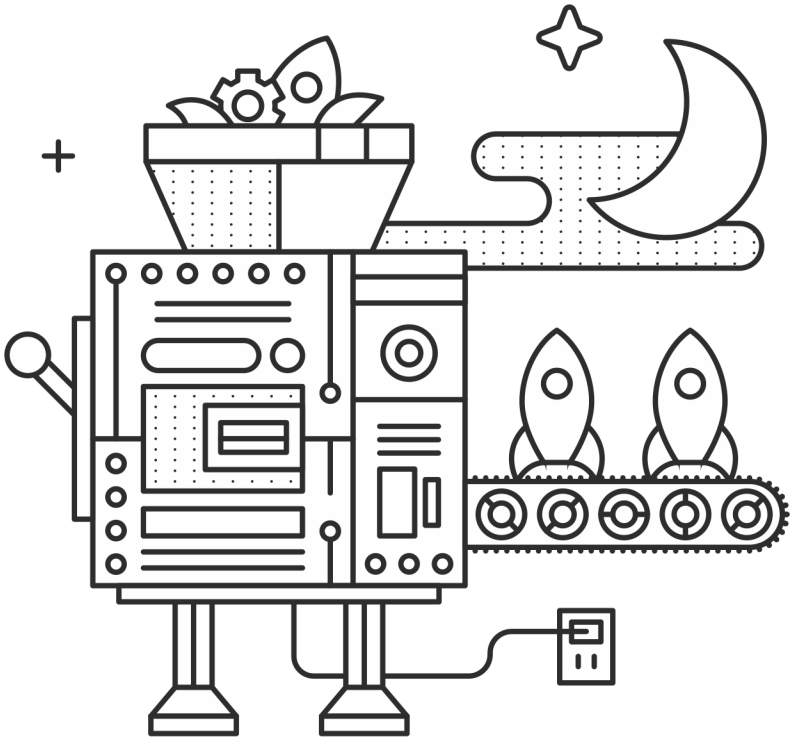
Now our application is far more modular than it was with one massive *views.py* file. The route definitions are simpler because we can group them together into blueprints and configure things like subdomains and URL prefixes once for each blueprint.

Summary

- A blueprint is a collection of views, templates, static files and other extensions that can be applied to an application.
- Blueprints are a great way to organize your application.
- In a divisional structure, each blueprint is a collection of views, templates and static files which constitute a particular section of your application.
- In a functional structure, each blueprint is just a collection of views. The templates are all kept together, as are the static files.
- To use a blueprint, you define it then register it on the application by calling `Flask.register_blueprint()`..
- You can define a dynamic URL prefix that will be applied to all routes in a blueprint.

- You can also define a dynamic subdomain for all routes in a blueprint.
- Refactoring a growing application to use blueprints can be done in five relatively small steps.

Templates



While Flask doesn't force us to use any particular templating language, it assumes that we're going to use Jinja. Most of the developers in the Flask community use Jinja, and I recommend that you do the same. There are a few extensions that have been written to let us use other templating languages, like Flask-Genshi and Flask-Mako. Stick with the default unless you have a good reason to use something else. Not knowing the Jinja syntax yet is not a good reason! You'll save yourself a lot of time and headache.

Note: Almost all resources imply Jinja2 when they refer to "Jinja." There was a Jinja1, but we won't be dealing with it here. When you see Jinja, we're talking about this: http://jinja.pocoo.org/

A quick primer on Jinja

The Jinja documentation does a great job of explaining the syntax and features of the language. I won't reiterate it all here, but I do want to make sure that you see this important note:

> There are two kinds of delimiters. `{% ... %}` and `{{ ... }}`. The first one is used to execute statements such as for-loops or assign values, the latter prints the result of the expression to the template.
>
> —Jinja Template Designer Documentation

How to organize templates

So where do templates fit into our app? If you've been following along at home, you may have noticed that Flask is really flexible about where we put things. Templates are no exception. You may also notice that there's usually a recommended place to put things. Two points for you. For templates, that place is in the package directory.

```
myapp/
    __init__.py
    models.py
    views/
    templates/
    static/
run.py
requirements.txt
```

```
templates/
    layout.html
    index.html
    about.html
    profile/
        layout.html
        index.html
    photos.html
    admin/
        layout.html
        index.html
        analytics.html
```

The structure of the *templates* directory parallels the structure of our routes. The template for the route *myapp.com/admin/analytics* is *templates/admin/analytics.html*. There are also some extra templates in there that won't be rendered directly. The *layout.html* files are meant to be inherited by the other templates.

Inheritance

Much like Batman's backstory, a well organized templates directory relies heavily on inheritance. The **parent template** usually defines a generalized structure that all of the **child templates** will work within. In our example, *layout.html* is a parent template and the other *.html* files are child templates.

You'll generally have one top-level *layout.html* that defines the general layout for your application and one for each section of your site. If you take a look at the directory above, you'll see that there is a top-level *myapp/templates/layout.html* as well as *myapp/templates/profile/layout.html* and *myapp/templates/admin/layout.html*. The last two files inherit and modify the first.

Inheritance is implemented with the `{% extends %}` and `{% block %}` tags. In the parent template, we can define blocks which will be populated by child templates.

```
{# _myapp/templates/layout.html_ #}

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{% block title %}{% endblock %}</title>
```

```
    </head>
    <body>
    {% block body %}
        <h1>This heading is defined in the parent.</h1>
    {% endblock %}
    </body>
</html>
```

In the child template, we can extend the parent template and define the contents of those blocks.

```
{# _myapp/templates/index.html_ #}

{% extends "layout.html" %}
{% block title %}Hello world!{% endblock %}
{% block body %}
    {{ super() }}
    <h2>This heading is defined in the child.</h2>
{% endblock %}
```

The `super()` function lets us include whatever was inside the block in the parent template.

Note: For more information on inheritance, refer to the Jinja Template Inheritence documentation.

Creating macros

We can implement DRY (Don't Repeat Yourself) principles in our templates by abstracting snippets of code that appear over and over into **macros**. If we're working on some HTML for our app's navigation, we might want to give a different class to the "active" link (i.e. the link to the current page). Without macros we'd end up with a block of `if ... else` statements that check each link to find the active one.

Macros provide a way to modularize that code; they work like functions. Let's look at how we'd mark the active link using a macro.

```
{# myapp/templates/layout.html #}

{% from "macros.html" import nav_link with context %}
<!DOCTYPE html>
<html lang="en">
    <head>
    {% block head %}
        <title>My application</title>
    {% endblock %}
    </head>
    <body>
        <ul class="nav-list">
            {{ nav_link('home', 'Home') }}
            {{ nav_link('about', 'About') }}
            {{ nav_link('contact', 'Get in touch') }}
        </ul>
    {% block body %}
    {% endblock %}
    </body>
</html>
```

What we are doing in this template is calling an undefined macro — `nav_link` — and passing it two parameters: the target endpoint (i.e. the function name for the target view) and the text we want to show.

Note: You may notice that we specified `with context` in the import statement. The Jinja **context** consists of the arguments passed to the `render_template()` function as well as the Jinja environment context from our Python code. These variables are made available in the template that is being rendered.

Some variables are explicitly passed by us, e.g. `render_template("index.html", color="red")`, but there are several variables and functions that Flask automatically includes in the context, e.g. `request`, `g` and `session`. When we say `{% from ... import ... with context %}` we are telling Jinja to make all of these variables available to the macro as well.

Note:

- All of the global variables that are passed to the Jinja context by Flask: http://flask.pocoo.org/docs/templating/#standard-context}
- We can define variables and functions that we want to be merged into the Jinja context with context processors: http://flask.pocoo.org/docs/templating/#context-processors

Now it's time to define the `nav_link` macro that we used in our template.

```
{# myapp/templates/macros.html #}

{% macro nav_link(endpoint, text) %}
{% if request.endpoint.endswith(endpoint) %}
    <li class="active"><a href="{{ url_for(endpoint) }}">{{text}}</a></li>
{% else %}
    <li><a href="{{ url_for(endpoint) }}">{{text}}</a></li>
{% endif %}
{% endmacro %}
```

Now we've defined the macro in *myapp/templates/macros.html*. In this macro we're using Flask's `request` object — which is available in the Jinja context by default — to check whether or not the current request was routed to the endpoint passed to `nav_link`. If it was, than we're currently on that page, and we can mark it as active.

Note: The from x import y statement takes a relative path for x. If our template was in *myapp/templates/user/blog.html* we would use `from "../macros.html" import nav_link with context`.

Custom filters

Jinja filters are functions that can be applied to the result of an expression in the `{{ ... }}` delimiters. It is applied before that result is printed to the template.

```
<h2>{{ article.title|title }}</h2>
```

In this code, the `title` filter will take `article.title` and return a title-cased version, which will then be printed to the template. This looks and works a lot like the UNIX practice of "piping" the output of one program to another.

Note: There are loads of built-in filters like `title`. See the full list in the Jinja docs.

We can define our own filters for use in our Jinja templates. As an example, we'll implement a simple `caps` filter to capitalize all of the letters in a string.

Note: Jinja already has an `upper` filter that does this, and a `capitalize` filter that capitalizes the first character and lowercases the rest. These also handle unicode conversion, but we'll keep our example simple to focus on the concept at hand.

We're going to define our filter in a module located at *myapp/util/filters.py*. This gives us a `util` package in which to put other miscellaneous modules.

```
# myapp/util/filters.py

from .. import app

@app.template_filter()
def caps(text):
    """Convert a string to all caps."""
    return text.uppercase()
```

In this code we are registering our function as a Jinja filter by using the `@app.template_filter()` decorator. The default filter name is just the name of the function, but you can pass an argument to the decorator to change that.

```
@app.template_filter('make_caps')
def caps(text):
    """Convert a string to all caps."""
    return text.uppercase()
```

Now we can call `make_caps` in the template rather than `caps`: `{{ "hello world!"|make_caps }}`.

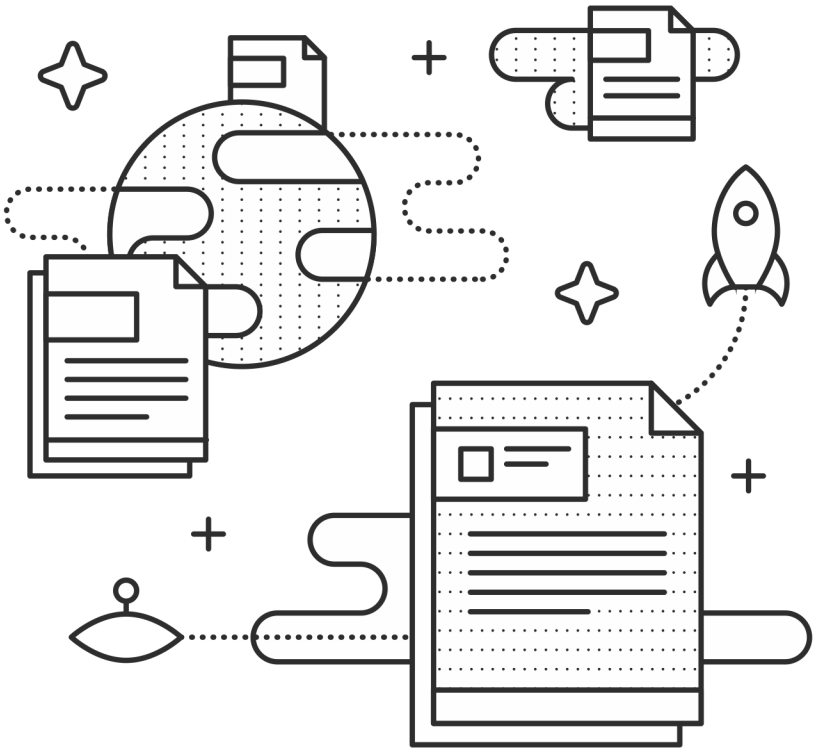
To make our filter available in the templates, we just need to import it in our top-level *__init.py__*.

```
# myapp/__init__.py

# Make sure app has been initialized first to prevent circular imports.
from .util import filters
```

Summary

- Use Jinja for templating.
- Jinja has two kinds of delimeters: `{% ... %}` and `{{ ... }}`. The first one is used to execute statements such as for-loops or assign values, the latter prints the result of the contained expression to the template.
- Templates should go in *myapp/templates/* — i.e. a directory inside of the application package.
- I recommend that the structure of the *templates/* directory mirror the URL structure of the app.
- You should have a top-level *layout.html* in *myapp/templates* as well as one for each section of the site. The former extend the latter.
- Macros are like functions made-up of template code.
- Filters are functions made-up of Python code and used in templates.

Static files



As their name suggests, static files are the files that don't change. In your average app, this includes CSS files, JavaScript files and images. They can also include audio files and other things of that nature.

Organizing your static files

We'll create a directory for our static files called *static* inside our application package.

```
myapp/
    __init__.py
    static/
    templates/
    views/
    models.py
run.py
```

How you organize the files in *static/* is a matter of personal preference. Personally, I get a little irked by having third-party libraries (e.g. jQuery, Bootstrap, etc.) mixed in with my own JavaScript and CSS files. To avoid this, I recommend separating third-party libraries out into a *lib/* folder within the appropriate directory. Some projects use *vendor/* instead of *lib/*.

```
static/
    css/
        lib/
            bootstrap.css
        style.css
        home.css
        admin.css
    js/
        lib/
            jquery.js
        home.js
        admin.js
    img/
        logo.svg
        favicon.ico
```

Serving a favicon

The files in our static directory will be served from *example.com/static/*. By default, web browsers and other software expects our favicon to be at *example.com/favicon.ico*. To fix this discrepency, we can add the following in the `<head>` section of our site template.

```
<link rel="shortcut icon"
    href="{{ url_for('static', filename='img/favicon.ico') }}">
```

Manage static assets with Flask-Assets

Flask-Assets is an extension for managing your static files. There are two really useful tools that Flask-Assets provides. First, it lets you define **bundles** of assets in your Python code that can be inserted together in your template. Second, it lets you **pre-process** those files. This means that you can combine and minify your CSS and JavaScript files so that the user only has to load two minified files (CSS and JavaScript) without forcing you to develop a complex asset pipeline. You can even compile your files from Sass, LESS, CoffeeScript and a bunch of other sources.

```
static/
    css/
        lib/
            reset.css
        common.css
        home.css
        admin.css
    js/
        lib/
            jquery-1.10.2.js
            Chart.js
        home.js
        admin.js
    img/
        logo.svg
        favicon.ico
```

Defining bundles

Our app has two sections: the public site and the admin panel, referred to as “home” and “admin” respectively in our app. We’ll define four bundles to cover this: a JavaScript and CSS bundle for each section. We’ll put these in an assets

module inside our `util` package.

```
# myapp/util/assets.py

from flask_assets import Bundle, Environment
from .. import app

bundles = {

    'home_js': Bundle(
        'js/lib/jquery-1.10.2.js',
        'js/home.js',
        output='gen/home.js'),

    'home_css': Bundle(
        'css/lib/reset.css',
        'css/common.css',
        'css/home.css',
        output='gen/home.css'),

    'admin_js': Bundle(
        'js/lib/jquery-1.10.2.js',
        'js/lib/Chart.js',
        'js/admin.js',
        output='gen/admin.js'),

    'admin_css': Bundle(
        'css/lib/reset.css',
        'css/common.css',
        'css/admin.css',
        output='gen/admin.css')
}

assets = Environment(app)

assets.register(bundles)
```

Flask-Assets combines your files in the order in which they are listed here. If *admin.js* requires *jquery-1.10.2.js*, make sure jquery is listed first.

We're defining the bundles in a dictionary to make it easy to register them. webassets, the package behind Flask-Assets lets us register bundles in a number of ways, including passing a dictionary like the one we made in this snippet. [1]

Since we're registering our bundles in `util.assets`, all we have to do is import that module in *__init__.py* after our app has been initialized.

```
# myapp/__init__.py

# [...] Initialize the app

from .util import assets
```

Using our bundles

To use our admin bundles, we'll insert them into the parent template for the admin section: *admin/layout.html*.

[1] We can see how bundle registration works in the source.

```
templates/
    home/
        layout.html
        index.html
        about.html
    admin/
        layout.html
        dash.html
        stats.html
```

```
{# myapp/templates/admin/layout.html #}

<!DOCTYPE html>
<html lang="en">
    <head>
        {% assets "admin_js" %}
            <script type="text/javascript" src="{{ ASSET_URL }}"></script>
        {% endassets %}
        {% assets "admin_css" %}
            <link rel="stylesheet" href="{{ ASSET_URL }}" />
        {% endassets %}
    </head>
    <body>
    {% block body %}
    {% endblock %}
    </body>
</html>
```

We can do the same thing for the home bundles in *templates/home/layout.html*.

Using filters

We can use filters to pre-process our static files. This is especially handy for minifying our JavaScript and CSS bundles.

```
# myapp/util/assets.py

# [...]

bundles = {

    'home_js': Bundle(
        'lib/jquery-1.10.2.js',
        'js/home.js',
        output='gen/home.js',
        filters='jsmin'),

    'home_css': Bundle(
        'lib/reset.css',
        'css/common.css',
        'css/home.css',
        output='gen/home.css',
        filters='cssmin'),

    'admin_js': Bundle(
        'lib/jquery-1.10.2.js',
        'lib/Chart.js',
        'js/admin.js',
        output='gen/admin.js',
```

```
        filters='jsmin'),

    'admin_css': Bundle(
        'lib/reset.css',
        'css/common.css',
        'css/admin.css',
        output='gen/admin.css',
        filters='cssmin')
}

# [...]
```

Note: To use the `jsmin` and `cssmin` filters, you'll need to install the `jsmin` and `cssmin` packages (e.g. with `pip install jsmin cssmin`). Make sure to add them to *requirements.txt* too.

Flask-Assets will merge and compress our files the first time the template is rendered, and it'll automatically update the compressed file when one of the source files changes.

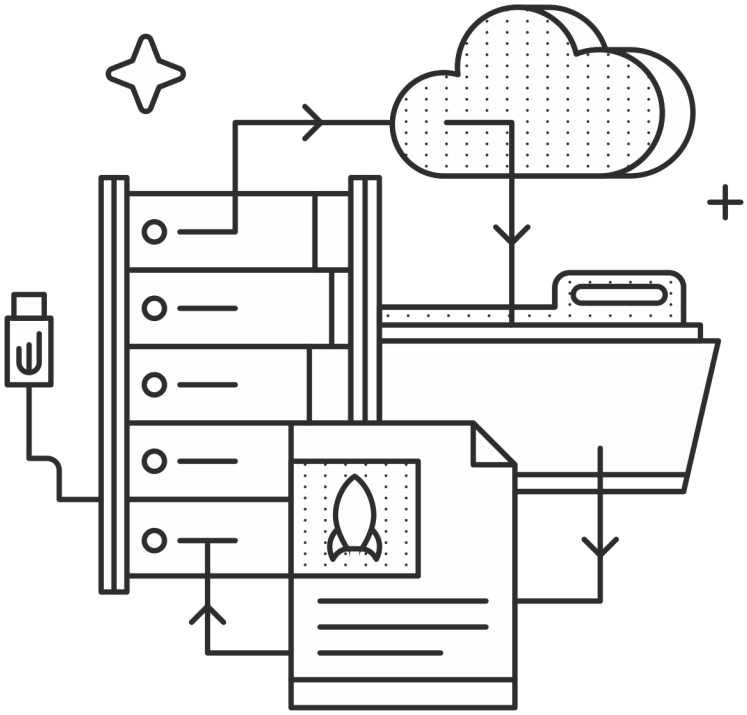
Note: If you set *ASSETS_DEBUG = True* in your config, Flask-Assets will output each source file individually instead of merging them.

Note: Take a look at some of the other filters that we can use with Flask-Assets.

Summary

- Static files go in the *static/* directory.
- Separate third-party libraries from your own static files.
- Specify the location of your favicon in your templates.
- Use Flask-Assets to insert static files in your templates.
- Flask-Assets can compile, combine and compress your static files.

Storing data



Most Flask applications are going to deal with storing data at some point. There are many different ways to store data. Finding the best one depends entirely on the data you are going to store. If you are storing relational data (e.g. a user has posts, posts have a user, etc.) a relational database is probably going to be the way to go (big suprise). Other types of data might be more suited to NoSQL data stores, such as MongoDB.

I'm not going to tell you how to choose a database engine for your application. There are people who will tell you that NoSQL is the only way to go and those who will say the same about relational databases. All I will say on that subject is that if you are unsure, a relational database (MySQL, PostgreSQL, etc.) will almost certainly work for whatever you're doing.

Plus, when you use a relational database you get to use SQLAlchemy and SQLAlchemy is fun.

SQLAlchemy

SQLAlchemy is an ORM (Object Relational Mapper). It's basically an abstraction layer that sits on top of the raw SQL queries being executed on our database. It provides a consistent API to a long list of database engines. The most popular include MySQL, PostgreSQL and SQLite. This makes it easy to move data between our models and our database and it makes it really easy to do other things like switch database engines and migrate our schemas.

There is a great Flask extension that makes using SQLAlchemy in Flask even easier. It's called Flask-SQLAlchemy. Flask-SQLAlchemy configures a lot of sane defaults for SQLAlchemy. It also handles some session management so

we don't have to deal with janitorial stuff in our application code.

Let's dive into some code. We're going to define some models then configure some SQLAlchemy. The models are going to go in *myapp/models.py*, but first we are going to define our database in *myapp/__init__.py*

```
# ourapp/__init__.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__, instance_relative_config=True)

app.config.from_object('config')
app.config.from_pyfile('config.py')

db = SQLAlchemy(app)
```

First we initialize and configure our Flask app and then we use it to initialize our SQLAlchemy database handler. We're going to use an instance folder for our database configuration so we should use the `instance_relative_config` option when initializing the app and then call `app.config.from_pyfile` to load it. Then we can define our models.

```
# ourapp/models.py

from . import db

class Engine(db.Model):

    # Columns

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)

    title = db.Column(db.String(128))

    thrust = db.Column(db.Integer, default=0)
```

`Column`, `Integer`, `String`, `Model` and other SQLAlchemy classes are all available via the `db` object constructed from Flask-SQLAlchemy. We have defined a model to store the current state of our spacecraft's engines. Each engine has an id, a title and a thrust level.

We still need to add some database information to our configuration. We're using an instance folder to keep confidential configuration variables out of version control, so we are going to put it in *instance/config.py*.

```
# instance/config.py

SQLALCHEMY_DATABASE_URI = "postgresql://user:password@localhost/spaceshipDB"
```

Note: Your database URI will be different depending on the engine you use and where it's hosted. See the SQLAlchemy documentation for this syntax.

Initializing the database

Now that the database is configured and we have defined a model, we can initialize the database. This step basically involves creating the database schema from the model definitions.

Normally that process might be a pain in the ... neck. Lucky for us, SQLAlchemy has a really cool command that will do all of this for us.

Let's open up a Python terminal in our repository root.

```
$ pwd
/Users/me/Code/myapp
$ workon myapp
(myapp)$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from myapp import db
>>> db.create_all()
>>>
```

Now, thanks to SQLAlchemy, our tables have been created in the database specified in our configuration.

Alembic migrations

The schema of a database is not set in stone. For example, we may want to add a `last_fired` column to the engine table. If we don't have any data, we can just update the model and run `db.create_all()` again. However, if we have six months of engine data logged in that table, we probably don't want to start over from scratch. That's where database migrations come in.

Alembic is a database migration tool created specifically for use with SQLAlchemy. It lets us keep a versioned history of our database schema so that we can later upgrade to a new schema and even downgrade back to an older one.

Alembic has an extensive tutorial to get you started, so I'll just give you a quick overview and point out a couple of things to watch out for.

We'll create our alembic "migration environment" via the `alembic init` command. Once we run this in our repository root we'll have a new directory with the very creative name *alembic*. Our repository will end up looking something like the example in this listing, adapted from the Alembic tutorial.

```
ourapp/
    alembic.ini
    alembic/
        env.py
        README
        script.py.mako
        versions/
            3512b954651e_add_account.py
            2b1ae634e5cd_add_order_id.py
            3adcc9a56557_rename_username_field.py
    myapp/
        __init__.py
        views.py
        models.py
        templates/
    run.py
    config.py
    requirements.txt
```

The *alembic/* directory has the scripts that migrate our data between versions. There is also an *alembic.ini* file that contains configuration information.

Note: Add *alembic.ini* to *.gitignore*! You are going to have your database credentials in this file, so you **do not** want it to end up in version control.

You do want to keep *alembic/* in version control though. It does not contain sensitive information (that can't already be derived from your source code) and keeping it in version control will mean having multiple copies should something happen to the files on your computer.

When it comes time to make a schema change, there are a couple of steps. First we run `alembic revision` to generate a migration script. Then we'll open up the newly generated Python file in *myapp/alembic/versions/* and fill in the `upgrade` and `downgrade` functions using the tools provided by Alembic's `op` object.

Once we have our migration script ready, we can run `alembic upgrade head` to migrade our data to the latest version.

Note: For the details on configuring Alembic, creating your migration scripts and running your migrations, see the Alembic tutorial.

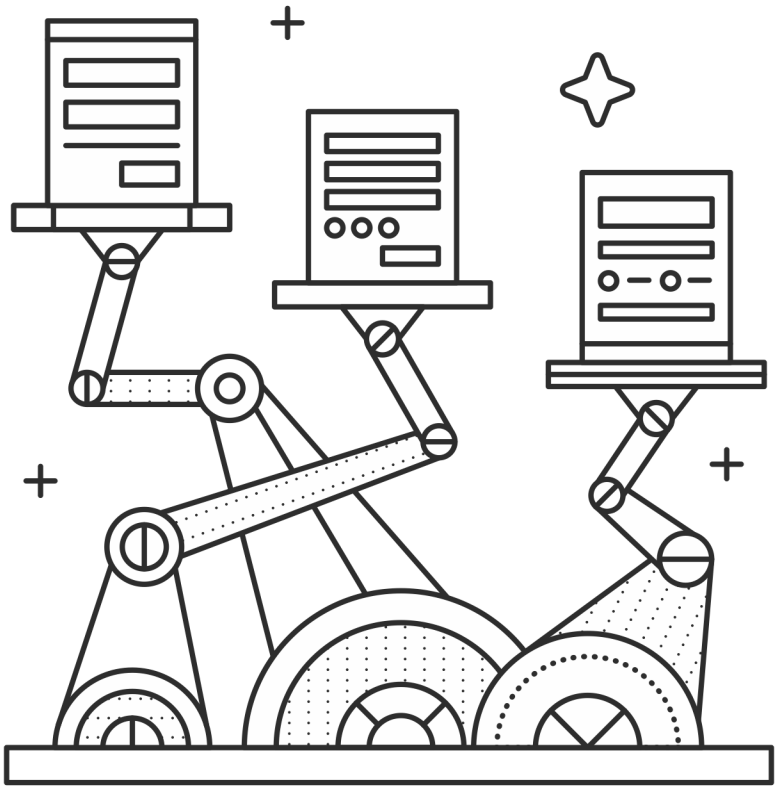
Warning: Don't forget to put a plan in place to back up your data. The details of that plan are outside the scope of this book, but you should always have your database backed up in a secure and robust way.

Note: The NoSQL scene is less established with Flask, but as long as the database engine of your choice has a Python library, you should be able to use it. There are even several extensions in the Flask extension registry to help integrate NoSQL engines with Flask.

Summary

- Use SQLAlchemy to work with relational databases.
- Use Flask-SQLAlchemy to work with SQLAlchemy.
- Alembic helps you migrate your data between schema changes.
- You can use NoSQL databases with Flask, but the methods and tools vary between engines.
- Back up your data!

Handling forms



The form is the basic element that lets users interact with our web application. Flask alone doesn't do anything to help us handle forms, but the Flask-WTF extension lets us use the popular WTForms package in our Flask applications. This package makes defining forms and handling submissions easy.

Flask-WTF

The first thing we want to do with Flask-WTF (after installing it) is to define a form in a `myapp.forms` package.

```
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
```

Note: Until version 0.9, Flask-WTF provided its own wrappers around the WTForms fields and validators. You may see a lot of code out in the wild that imports `TextField`, `PasswordField`, etc. from `flask_wtforms` instead

of `wtforms`.

As of 0.9, we should be importing that stuff straight from `wtforms`.

The form we defined is going to be a user sign-in form. We could have called it `SignInForm()`, but by keeping things a little more abstract, we can re-use this same form class for other things, like a sign-up form. If we were to define purpose-specific form classes we'd end up with a lot of identical forms for no good reason. It's much cleaner to name forms based on the fields they contain, as that is what makes them unique. Of course, sometimes we'll have long, one-off forms that we might want to give a more context-specific name.

This sign-in form can do a few of things for us. It can secure our app against CSRF vulnerabilites, validate user input and render the appropriate markup for whatever fields we define for it.

CSRF Protection and validation

CSRF stands for cross site request forgery. CSRF attacks involve a third party forging a request (like a form submission) to an app's server. A vulnerable server assumes that the data is coming from a form on its own site and takes action accordingly.

As an example, let's say that an email provider lets you delete your account by submitting a form. The form sends a POST request to an `account_delete` endpoint on their server and deletes the account that was logged-in when the form was submitted. We can create a form on our own site that sends a POST request to the same `account_delete` endpoint. Now, if we can get someone to click 'submit' on our form (or do it via JavaScript when they load the page) their logged-in account with the email provider will be deleted. Unless of course the email provider knows not to assume that form submissions are coming from their own forms.

So how do we stop assuming that POST requests come from our own forms? WTForms makes it possible by generating a unique token when rendering each form. That token is meant to be passed back to the server, along with the form data in the POST request and must be validated before the form is accepted. The key is that the token is tied to a value stored in the user's session (cookies) and expires after a certain amount of time (30 minutes by default). This way the only person who can submit a valid form is the person who loaded the page (or at least someone at the same computer), and they can only do it for 30 minutes after loading the page.

Note:

- Read more on how WTForms generates these tokens in the docs.
- Learn about CSRF in the OWASP wiki.

To start using Flask-WTF for CSRF protection, we'll need to define a view for our login page.

```
# ourapp/views.py

from flask import render_template, redirect, url_for

from . import app
from .forms import EmailPasswordForm

@app.route('/login', methods=["GET", "POST"])
def login():
    form = EmailPasswordForm()
    if form.validate_on_submit():

        # Check the password and log the user in
        # [...]

```

```
        return redirect(url_for('index'))
    return render_template('login.html', form=form)
```

We import our form from our `forms` package and instantiate it in the view. Then we run `form.validate_on_submit()`. This function returns `True` if the form has been both submitted (i.e. if the HTTP method is PUT or POST) and validated by the validators we defined in *forms.py*.

Note:

- Documentation for Form.validate_on_submit
- Source for Form.validate_on_submit

If the form has been submitted and validated, we can continue with the login logic. If it hasn't been submitted (i.e. it's just a GET request), we want to pass the form object to our template so it can be rendered. Here's what the template looks like when we're using CSRF protection.

```
{# ourapp/templates/login.html #}

{% extends "layout.html" %}
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form action="{{ url_for('login') }}" method="post">
            <input type="text" name="email" />
            <input type="password" name="password" />
            {{ form.csrf_token }}
        </form>
    </body>
</html>
```

`{{ form.csrf_token }}` renders a hidden field containing one of those fancy CSRF tokens and WTForms looks for that field when it validates the form. We don't have to worry about including any special "is the token valid" logic. Hooray!

Protecting AJAX calls with CSRF tokens

Flask-WTF CSRF tokens aren't limited to protecting form submissions. If your app makes other requests that might be forged (especially AJAX calls) you can add CSRF protection there too!

Note: The Flask-WTF documentation talks more about using these CSRF tokens in AJAX calls.

Custom validators

In addition to the built-in form validators provided by WTForms (e.g. `Required()`, `Email()`, etc.), we can create our own validators. We'll demonstrate this by making a `Unique()` validator that will check a database and make sure that the value provided by the user doesn't already exist. This could be used to make sure that a username or email address isn't already in use. Without WTForms, we'd probably be doing these checks in the view, but now we can abstract that away to the form itself.

We'll start by defining a simple sign-up form.

```python
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
```

Now we want to add our validator to make sure that the email they provide isn't already in the database. We'll put the validator in a new `util` module, `util.validators`.

```python
# ourapp/util/validators.py
from wtforms.validators import ValidationError

class Unique(object):
    def __init__(self, model, field, message=u'This element already exists.'):
        self.model = model
        self.field = field

    def __call__(self, form, field):
        check = self.model.query.filter(self.field == field.data).first()
        if check:
            raise ValidationError(self.message)
```

This validator assumes that we're using SQLAlchemy to define our models. WTForms expects validators to return some sort of callable (e.g. a callable class).

In *__init__.py* we can specify which arguments should be passed to the validator. In this case we want to pass the relevant model (e.g. the `User` model in our case) and the field to check. When the validator is called, it will raise a `ValidationError` if any instance of the defined model matches the value submitted in the form. We've also made it possible to add a message with a generic default that will be included in the `ValidationError`.

Now we can modify `EmailPasswordForm` to use the `Unique` validator.

```python
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired

from .util.validators import Unique
from .models import User

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[DataRequired(), Email(),
        Unique(
            User,
            User.email,
            message='There is already an account with that email.'])
    password = PasswordField('Password', validators=[DataRequired()])
```

Note: Our validator doesn't have to be a callable class. It could also be a factory that returns a callable or just a callable directly. The WTForms documentation has some examples.

Rendering forms

WTForms can also help us render the HTML for the forms. The `Field` class implemented by WTForms renders an HTML representation of that field, so we just have to call the form fields to render them in our template. It's just like rendering the `csrf_token` field. Listing gives an example of a login template using WTForms to render our fields.

```
{# ourapp/templates/login.html #}

{% extends "layout.html" %}
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form action="" method="post">
            {{ form.email }}
            {{ form.password }}
            {{ form.csrf_token }}
        </form>
    </body>
</html>
```

We can customize how the fields are rendered by passing field properties as arguments to the call.

```
<form action="" method="post">
    {{ form.email.label }}: {{ form.email(placeholder='yourname@email.com') }}
    <br>
    {{ form.password.label }}: {{ form.password }}
    <br>
    {{ form.csrf_token }}
</form>
```

Note: If we want to pass the "class" HTML attribute, we have to use `class_=''` since "class" is a reserved keyword in Python.

Note: The WTForms documentation has a list of available field properties.

Note: You may notice that we don't need to use Jinja's `|safe` filter. This is because WTForms renders HTML safe strings.

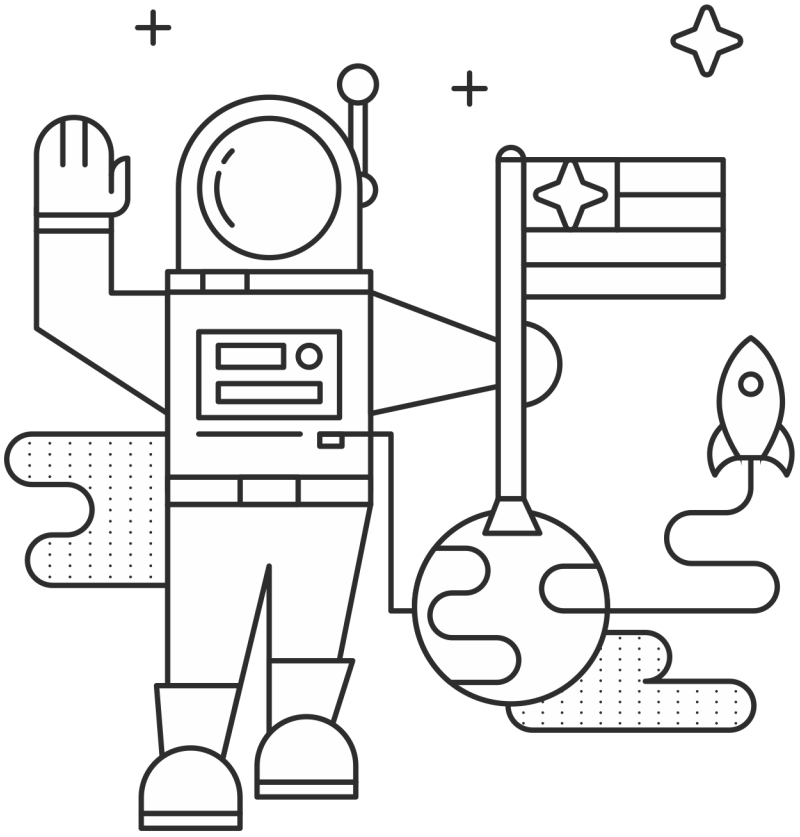
Read more in the documentation.

Summary

- Forms can be scary from a security perspective.
- WTForms (and Flask-WTF) make it easy to define, secure and render your forms.
- Use the CSRF protection provided by Flask-WTF to secure your forms.
- You can use Flask-WTF to protect AJAX calls against CSRF attacks too.
- Define custom form validators to keep validation logic out of your views.

- Use the WTForms field rendering to render your form's HTML so you don't have to update it every time you make some changes to the form definition.

Patterns for handling users



One of the most common things that modern web applications need to do is handle users. An application with basic account features needs to handle a lot of things like registration, email confirmation, securely storing passwords, secure password reset, authentication and more. Since a lot of security issues present themselves when it comes to handling users, it's generally best to stick to standard patterns in this area.

Note: In this chapter I'm going to assume that you're using SQLAlchemy models and WTForms to handle your form input. If you aren't using those, you'll need to adapt these patterns to your preferred methods.

Email confirmation

When a new user gives us their email, we generally want to confirm that they gave us the right one. Once we've made that confirmation, we can confidently send password reset links and other sensitive information to our users without wondering who is on the receiving end.

One of the most common patterns for confirming emails is to send a password reset link with a unique URL that, when visited, confirms that user's email address. For example, john@gmail.com signs up at our application. We register him in the database with an `email_confirmed` column set to `False` and fire off an email to john@gmail.com with a unique URL. This URL usually contains a unique token, e.g. *http://myapp.com/accounts/confirm/Q2hhZCBDYXRsZXR0IHJvY2tzIG15IHNvY2tz*. When John gets that email, he clicks the link. Our app sees the token, knows which email to confirm and sets John's `email_confirmed` column to `True`.

How do we know which email to confirm with a given token? One way would be to store the token in the database when it is created and check that table when we receive the confirmation request. That's a lot of overhead and, lucky for us, it's unnecessary.

We're going to encode the email address in the token. The token will also contain a timestamp to let us set a time limit on how long it's valid. To do this, we'll use the `itsdangerous` package. This package gives us tools to send sensitive data into untrusted environments (like sending an email confirmation token to an unconfirmed email). In this case, we're going to use an instance of the `URLSafeTimedSerializer` class.

```
# ourapp/util/security.py

from itsdangerous import URLSafeTimedSerializer

from .. import app

ts = URLSafeTimedSerializer(app.config["SECRET_KEY"])
```

We can use that serializer to generate a confirmation token when a user gives us their email address. We'll implement a simple account creation process using this method.

```
# ourapp/views.py

from flask import redirect, render_template, url_for

from . import app, db
from .forms import EmailPasswordForm
from .util import ts, send_email

@app.route('/accounts/create', methods=["GET", "POST"])
def create_account():
    form = EmailPasswordForm()
    if form.validate_on_submit():
        user = User(
            email = form.email.data,
            password = form.password.data
        )
        db.session.add(user)
        db.session.commit()

        # Now we'll send the email confirmation link
        subject = "Confirm your email"

        token = ts.dumps(self.email, salt='email-confirm-key')

        confirm_url = url_for(
            'confirm_email',
            token=token,
            _external=True)

        html = render_template(
            'email/activate.html',
```

```
            confirm_url=confirm_url)

        # We'll assume that send_email has been defined in myapp/util.py
        send_email(user.email, subject, html)

        return redirect(url_for("index"))

    return render_template("accounts/create.html", form=form)
```

The view that we've defined handles the creation of the user and sends off an email to the given email address. You may notice that we're using a template to generate the HTML for the email.

```
{# ourapp/templates/email/activate.html #}

Your account was successfully created. Please click the link below<br>
to confirm your email address and activate your account:

<p>
<a href="{{ confirm_url }}">{{ confirm_url }}</a>
</p>

<p>
--<br>
Questions? Comments? Email hello@myapp.com.
</p>
```

Okay, so now we just need to implement a view that handles the confirmation link in that email.

```
# ourapp/views.py

@app.route('/confirm/<token>')
def confirm_email(token):
    try:
        email = ts.loads(token, salt="email-confirm-key", max_age=86400)
    except:
        abort(404)

    user = User.query.filter_by(email=email).first_or_404()

    user.email_confirmed = True

    db.session.add(user)
    db.session.commit()

    return redirect(url_for('signin'))
```

This view is a simple form view. We just add the `try ... except` bit at the beginning to check that the token is valid. The token contains a timestamp, so we can tell `ts.loads()` to raise an exception if it is older than `max_age`. In this case, we're setting `max_age` to 86400 seconds, i.e. 24 hours.

Note: You can use very similar methods to implement an email update feature. Just send a confirmation link to the new email address with a token that contains both the old and the new addresses. If the token is valid, update the old address with the new one.

Storing passwords

Rule number one of handling users is to hash passwords with the Bcrypt (or scrypt, but we'll use Bcrypt here) algorithm before storing them. We never store passwords in plain text. It's a massive security issue and it's unfair to our users. All of the hard work has already been done and abstracted away for us, so there's no excuse for not following the best practices here.

Note: OWASP is one of the industry's most trusted source for information regarding web application security. Take a look at some of their recommendations for secure coding.

We'll go ahead and use the Flask-Bcrypt extension to implement the bcrypt package in our application. This extension is basically just a wrapper around the `py-bcrypt` package, but it does handle a few things that would be annoying to do ourselves (like checking string encodings before comparing hashes).

```
# ourapp/__init__.py

from flask_bcrypt import Bcrypt

bcrypt = Bcrypt(app)
```

One of the reasons that the Bcrypt algorithm is so highly recommended is that it is "future adaptable." This means that over time, as computing power becomes cheaper, we can make it more and more difficult to brute force the hash by guessing millions of possible passwords. The more "rounds" we use to hash the password, the longer it will take to make one guess. If we hash our passwords 20 times with the algorithm before storing them the attacker has to hash each of their guesses 20 times.

Keep in mind that if we're hashing our passwords 20 times then our application is going to take a long time to return a response that depends on that process completing. This means that when choosing the number of rounds to use, we have to balance security and usability. The number of rounds we can complete in a given amount of time will depend on the computational resources available to our application. It's a good idea to test out some different numbers and shoot for between 0.25 and 0.5 seconds to hash a password. We should try to use at least 12 rounds though.

To test the time it takes to hash a password, we can time a quick Python script that, well, hashes a password.

```
# benchmark.py

from flask_bcrypt import generate_password_hash

# Change the number of rounds (second argument) until it takes between
# 0.25 and 0.5 seconds to run.
generate_password_hash('password1', 12)
```

Now we can keep timing our changes to the number of rounds with the UNIX `time` utility.

```
$ time python test.py

real    0m0.496s
user    0m0.464s
sys     0m0.024s
```

I did a quick benchmark on a small server that I have handy and 12 rounds seemed to take the right amount of time, so I'll configure our example to use that.

```
# config.py

BCRYPT_LOG_ROUNDS = 12
```

Now that Flask-Bcrypt is configured, it's time to start hashing passwords. We could do this manually in the view that receives the request from the sign-up form, but we'd have to do it again in the password reset and password change views. Instead, what we'll do is abstract away the hashing so that our app does it without us even thinking about it. We'll use a **setter** so that when we set `user.password = 'password1'`, it's automatically hashed with Bcrypt before being stored.

```
# ourapp/models.py

from sqlalchemy.ext.hybrid import hybrid_property

from . import bcrypt, db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(64), unique=True)
    _password = db.Column(db.String(128))

    @hybrid_property
    def password(self):
        return self._password

    @password.setter
    def _set_password(self, plaintext):
        self._password = bcrypt.generate_password_hash(plaintext)
```

We're using SQLAlchemy's hybrid extension to define a property with several different functions called from the same interface. Our setter is called when we assign a value to the `user.password` property. In it, we hash the plaintext password and store it in the `_password` column of the user table. Since we're using a hybrid property we can then access the hashed password via the same `user.password` property.

Now we can implement a sign-up view for an app using this model.

```
# ourapp/views.py

from . import app, db
from .forms import EmailPasswordForm
from .models import User

@app.route('/signup', methods=["GET", "POST"])
def signup():
    form = EmailPasswordForm()
    if form.validate_on_submit():
        user = User(username=form.username.data, password=form.password.data)
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('index'))

    return render_template('signup.html', form=form)
```

Authentication

Now that we've got a user in the database, we can implement authentication. We'll want to let a user submit a form with their username and password (though this might be email and password for some apps), then make sure that they gave us the correct password. If it all checks out, we'll mark them as authenticated by setting a cookie in their browser. The next time they make a request we'll know that they have already logged in by looking for that cookie.

Let's start by defining a `UsernamePassword` form with WTForms.

```python
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired


class UsernamePasswordForm(Form):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
```

Next we'll add a method to our user model that compares a string with the hashed password stored for that user.

```
# ourapp/models.py

from . import db

class User(db.Model):

    # [...] columns and properties

    def is_correct_password(self, plaintext)
        return bcrypt.check_password_hash(self._password, plaintext)
```

Flask-Login

Our next goal is to define a sign-in view that serves and accepts our form. If the user enters the correct credentials, we will authenticate them using the Flask-Login extension. This extension simplifies the process of handling user sessions and authentication.

We need to do a little bit of configuration to get Flask-Login ready to roll.

In *__init__.py* we'll define the Flask-Login `login_manager`.

```python
# ourapp/__init__.py

from flask_login import LoginManager

# Create and configure app
# [...]

from .models import User

login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view =  "signin"

@login_manager.user_loader
def load_user(userid):
    return User.query.filter(User.id==userid).first()
```

Here we created an instance of the `LoginManager`, initialized it with our `app` object, defined the login view and told it how to get a user object with a user's `id`. This is the baseline configuration we should have for Flask-Login.

Note: See more ways to customize Flask-Login.

Now we can define the `signin` view that will handle authentication.

```python
# ourapp/views.py

from flask import redirect, url_for

from flask_login import login_user

from . import app
from .forms import UsernamePasswordForm

@app.route('signin', methods=["GET", "POST"])
def signin():
    form = UsernamePasswordForm()

    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first_or_404()
        if user.is_correct_password(form.password.data):
            login_user(user)

            return redirect(url_for('index'))
        else:
            return redirect(url_for('signin'))
    return render_template('signin.html', form=form)
```

We simply import the `login_user` function from Flask-Login, check a user's login credentials and call `login_user(user)`. You can log the current user out with `logout_user()`.

```python
# ourapp/views.py

from flask import redirect, url_for
from flask_login import logout_user

from . import app

@app.route('/signout')
def signout():
    logout_user()

    return redirect(url_for('index'))
```

Forgot your password

We'll generally want to implement a "Forgot your password" feature that lets a user recover their account by email. This area has a plethora of potential vulnerabilities because the whole point is to let an unauthenticated user take over an account. We'll implement our password reset using some of the same techniques as our email confirmation.

We'll need a form to request a reset for a given account's email and a form to choose a new password once we've confirmed that the unauthenticated user has access to that email address. The code in this section assumes that our user model has an email and a password, where the password is a hybrid property as we previously created.

> **Warning:** Don't send password reset links to an unconfirmed email address! You want to be sure that you are sending this link to the right person.

We're going to need two forms. One is to request that a reset link be sent to a certain email and the other is to change the password once the email has been verified.

```
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email

class EmailForm(Form):
    email = TextField('Email', validators=[DataRequired(), Email()])

class PasswordForm(Form):
    password = PasswordField('Password', validators=[DataRequired()])
```

This code assumes that our password reset form just needs one field for the password. Many apps require the user to enter their new password twice to confirm that they haven't made a typo. To do this, we'd simply add another `PasswordField` and add the `EqualTo` WTForms validator to the main password field.

Note: There a lot of interesting discussions in the User Experience (UX) community about the best way to handle this in sign-up forms. I personally like the thoughts of one Stack Exchange user (Roger Attrill) who said:

"We should not ask for password twice - we should ask for it once and make sure that the 'forgot password' system works seamlessly and flawlessly."

- Read more about this topic in the thread on the User Experience Stack Exchange.
- There are also some cool ideas for simplifying sign-up and sign-in forms in an article on Smashing Magazine article.

Now we'll implement the first view of our process, where a user can request that a password reset link be sent for a given email address.

```
# ourapp/views.py

from flask import redirect, url_for, render_template

from . import app
from .forms import EmailForm
from .models import User
from .util import send_email, ts

@app.route('/reset', methods=["GET", "POST"])
def reset():
    form = EmailForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first_or_404()

        subject = "Password reset requested"

        # Here we use the URLSafeTimedSerializer we created in `util` at the
        # beginning of the chapter
        token = ts.dumps(user.email, salt='recover-key')

        recover_url = url_for(
            'reset_with_token',
            token=token,
            _external=True)

        html = render_template(
            'email/recover.html',
```

```
            recover_url=recover_url)

        # Let's assume that send_email was defined in myapp/util.py
        send_email(user.email, subject, html)

        return redirect(url_for('index'))
    return render_template('reset.html', form=form)
```

When the form receives an email address, we grab the user with that email address, generate a reset token and send them a password reset URL. That URL routes them to a view that will validate the token and let them reset the password.

```
# ourapp/views.py

from flask import redirect, url_for, render_template

from . import app, db
from .forms import PasswordForm
from .models import User
from .util import ts

@app.route('/reset/<token>', methods=["GET", "POST"])
def reset_with_token(token):
    try:
        email = ts.loads(token, salt="recover-key", max_age=86400)
    except:
        abort(404)

    form = PasswordForm()

    if form.validate_on_submit():
        user = User.query.filter_by(email=email).first_or_404()

        user.password = form.password.data

        db.session.add(user)
        db.session.commit()

        return redirect(url_for('signin'))

    return render_template('reset_with_token.html', form=form, token=token)
```

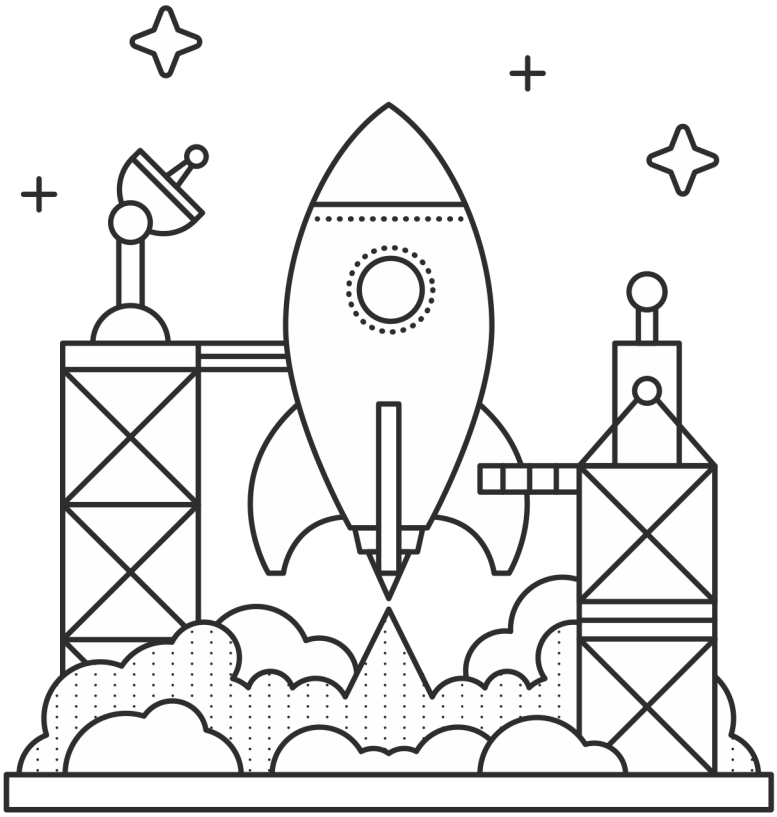
We're using the same token validation method as we did to confirm the user's email address. The view passes the token from the URL back into the template. Then the template uses the token to submit the form to the right URL. Let's have a look at what that template might look like.

```
{# ourapp/templates/reset_with_token.html #}

{% extends "layout.html" %}

{% block body %}
<form action="{{ url_for('reset_with_token', token=token) }}" method="POST">
    {{ form.password.label }}: {{ form.password }}<br>
    {{ form.csrf_token }}
    <input type="submit" value="Change my password" />
</form>
{% endblock %}
```

Summary

- Use the itsdangerous package to create and validate tokens sent to an email address.
- You can use these tokens to validate emails when a user creates an account, changes their email or forgets their password.
- Authenticate users using the Flask-Login extension to avoid dealing with a bunch of session management stuff yourself.
- Always think about how a malicious user could abuse your app to do things that you didn't intend.

Deployment



We're finally ready to show our app to the world. It's time to deploy. This process can be a pain because there are so many moving parts. There are a lot of choices to make when it comes to our production stack as well. In this chapter, we're going to talk about some of the important pieces and some of the options we have with each.

The Host

We're going to need a server somewhere. There are thousands of providers out there, but these are the three that I personally recommend. I'm not going to go over the details of how to get started with them, because that's out of the

scope of this book. Instead I'll talk about their benefits with regards to hosting Flask applications.

Amazon Web Services EC2

Amazon Web Services is a collection of services provided by ... Amazon! There's a good chance that you've heard of them before as they're probably the most popular choice for new startups these days. The AWS service that we're most concerned with here is EC2, or Elastic Compute Cloud. The big selling point of EC2 is that we get virtual servers - or **instances** as they're called in AWS parlance - that spin up in seconds. If we need to scale our app quickly it's just a matter of spinning up a few more EC2 instances for our app and sticking them behind a load balancer (we can even use the AWS Elastic Load Balancer).

With regards to Flask, AWS is just a regular old virtual server. We can spin it up with our favorite linux distro and install our Flask app and our server stack without much overhead. It means that we're going to need a certain amount of systems administration knowledge though.

Heroku

Heroku is an application hosting service that is built on top of AWS services like EC2. They let us take advantage of the convenience of EC2 without the requisite systems administration experience.

With Heroku, we deploy our application with a `git push` to their server. This is really convenient when we don't want to spend our time SSHing into a server, installing and configuring software and coming up with a sane deployment procedure. This convenience comes at a price of course, though both AWS and Heroku offer a certain amount of free service.

Note: Heroku has a tutorial on deploying Flask with their service.

Note: Administrating your own databases can be time consuming and doing it well requires some experience. It's great to learn about database administration by doing it yourself for your side projects, but sometimes you'd like to save time and effort by outsourcing that part to professionals.

Both Heroku and AWS have database management offerings. I don't have personal experience with either yet, but I've heard great things. It's worth considering if you want to make sure your data is being secured and backed-up without having to do it yourself.

- Heroku Postgres
- Amazon RDS

Digital Ocean

Digital Ocean is an EC2 competitor that has recently begun to take off. Like EC2, Digital Ocean lets us spin up virtual servers - now called **droplets** - quickly. All droplets run on SSDs, which isn't something we get at the lower levels of EC2. The biggest selling point for me personally is an interface that is far simpler and easier to use than the AWS control panel. Digital Ocean is my preference for hosting and I recommend that you take a look at them.

The Flask deployment experience on Digital Ocean is roughly the same as on EC2. We're starting with a clean linux distribution and installing our server stack from there.

Note: Digital Ocean was nice enough to make a contribution to the Kickstarter campaign for *Explore Flask*. With that said, I promise that my recommendation comes from my own experience as a user. If I didn't like them, I wouldn't

have asked them to pledge in the first place.

The stack

This section will cover some of the software that we'll need to install on our server to serve our Flask application to the world. The basic stack is a front server that reverse proxies requests to an application runner that is running our Flask app. We'll usually have a database too, so we'll talk a little about those options as well.

Application runner

The server that we use to run Flask locally when we're developing our application isn't good at handling real requests. When we're actually serving our application to the public, we want to run it with an application runner like Gunicorn. Gunicorn handles requests and takes care of complicated things like threading.

To use Gunicorn, we install the `gunicorn` package in our virtual environment with Pip. Running our app is a simple command away.

```
# rocket.py

from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
        return "Hello World!"
```

A fine app indeed. Now, to serve it up with Gunicorn, we simply run the `gunicorn` command.

```
(ourapp)$ gunicorn rocket:app
2014-03-19 16:28:54 [62924] [INFO] Starting gunicorn 18.0
2014-03-19 16:28:54 [62924] [INFO] Listening at: http://127.0.0.1:8000 (62924)
2014-03-19 16:28:54 [62924] [INFO] Using worker: sync
2014-03-19 16:28:54 [62927] [INFO] Booting worker with pid: 62927
```

At this point, we should see "Hello World!" when we navigate our browser to *http://127.0.0.1:8000*.

To run this server in the background (i.e. daemonize it), we can pass the `-D` option to Gunicorn. That way it'll run even after we close our current terminal session.

If we daemonize Gunicorn, we might have a hard time finding the process to close later when we want to stop the server. We can tell Gunicorn to stick the process ID in a file so that we can stop or restart it later without searching through lists of running processess. We use the `-p <file>` option to do that.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -D
(ourapp)$ cat rocket.pid
63101
```

To restart and kill the server, we can run `kill -HUP` and `kill` respectively.

```
(ourapp)$ kill -HUP `cat rocket.pid`
(ourapp)$ kill `cat rocket.pid`
```

By default Gunicorn runs on port 8000. We can change the port by adding the `-b` bind option.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -b 127.0.0.1:7999 -D
```

Making Gunicorn public

Warning: Gunicorn is meant to sit behind a reverse proxy. If you tell it to listen to requests coming in from the public, it makes an easy target for denial of service attacks. It's just not meant to handle those kinds of requests. Only allow outside connections for debugging purposes and make sure to switch it back to only allowing internal connections when you're done.

If we run Gunicorn like we have in the listings, we won't be able to access it from our local system. That's because Gunicorn binds to 127.0.0.1 by default. This means that it will only listen to connections coming from the server itself. This is the behavior that we want when we have a reverse proxy server that is sitting between the public and our Gunicorn server. If, however, we need to make requests from outside of the server for debugging purposes, we can tell Gunicorn to bind to 0.0.0.0. This tells it to listen for all requests.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -b 0.0.0.0:8000 -D
```

Note:

- Read more about running and deploying Gunicorn in the documentation.
- Fabric is a tool that lets you run all of these deployment and management commands from the comfort of your local machine without SSHing into every server.

Nginx Reverse Proxy

A reverse proxy handles public HTTP requests, sends them back to Gunicorn and gives the response back to the requesting client. Nginx can be used very effectively as a reverse proxy and Gunicorn "strongly advises" that we use it.

To configure Nginx as a reverse proxy to a Gunicorn server running on 127.0.0.1:8000, we can create a file for our app: */etc/nginx/sites-available/exploreflask.com*.

```
# /etc/nginx/sites-available/exploreflask.com

# Redirect www.exploreflask.com to exploreflask.com
server {
        server_name www.exploreflask.com;
        rewrite ^ http://exploreflask.com/ permanent;
}

# Handle requests to exploreflask.com on port 80
server {
        listen 80;
        server_name exploreflask.com;

                # Handle all locations
        location / {
                        # Pass the request to Gunicorn
                proxy_pass http://127.0.0.1:8000;

                # Set some HTTP headers so that our app knows where the
                # request really came from
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
        }
}
```

Now we'll create a symlink to this file at */etc/nginx/sites-enabled* and restart Nginx.

```
$ sudo ln -s \
/etc/nginx/sites-available/exploreflask.com \
/etc/nginx/sites-enabled/exploreflask.com
```

We should now be able to make our requests to Nginx and receive the response from our app.

Note: The Nginx configuration section in the Gunicorn docs will give you more information about setting Nginx up for this purpose.

ProxyFix

We may run into some issues with Flask not properly handling the proxied requests. It has to do with those headers we set in the Nginx configuration. We can use the Werkzeug ProxyFix to ... fix the proxy.

```
# app.py

from flask import Flask

# Import the fixer
from werkzeug.contrib.fixers import ProxyFix

app = Flask(__name__)

# Use the fixer
app.wsgi_app = ProxyFix(app.wsgi_app)

@app.route('/')
def index():
        return "Hello World!"
```

Note:

- Read more about ProxyFix in the Werkzeug docs.

Summary

- Three good choices for hosting Flask apps are AWS EC2, Heroku and Digital Ocean.
- The basic deployment stack for a Flask application consists of the app, an application runner like Gunicorn and a reverse proxy like Nginx.
- Gunicorn should sit behind Nginx and listen on 127.0.0.1 (internal requests) not 0.0.0.0 (external requests).
- Use Werkzeug's ProxyFix to handle the appropriate proxy headers in your Flask application.

Conclusion

I don't feel like there's a lot to conclude at this point. I hope reading this book has helped you in your adventure with Flask. If that's the case, please get in touch with me! I would love to hear from people who enjoyed reading this. Feel free to let me know if you have any suggestions to improve the book as well.

Thanks for reading!

- Robert

CHAPTER 3

Thank you

First of all, I'd like to say thank you to my volunteer editor, Will Kahn-Greene. He was great about going over my very rough drafts and helping me decide on the scope and direction of the content. I'm looking forward to working with him to manage the project into the future.

Another big thanks to everyone who took the time to talk with me about how they are using Flask. This includes Armin Ronacher (Flask creator), Mark Harviston (Elsevier), Glenn Yonemitsu (Markup Hive), Andy Parsons (Happify), Oleg Lavrovsky (Apps with love), Joel Anderson (Cloudmancer) and Mahmoud Abdelkader (Balanced).

The cover and all the illustrations in this book were done by Dominic Flask.

Explore Flask wouldn't be happening if it weren't for the hundreds of people who backed the project on Kickstarter. A big thanks to a particularly generous sponsor, Balanced Payments:

As promised in the Kickstarter project, here are the names of all of the generous men and women who pledged $50 or more:

CodeLesson, Sam Black, Michał Bartoszkiewicz, Chad Catlett, Jacob Kaplan-Moss, John Schrom, Zach White, Dorothy L. Erwin, Brandon Brown, Fredrik Larsson, Karsten Hoffrath (khoffrath), Jonathan Chen, Mitch Wainer, John Cleaver, Paul Baines, Brandon Bennett, Gaelan Adams, Nick Charlton, Dustin Chapman and Senko Rašić.

CHAPTER 4

License

In the spirit of open source software, I'm placing all of the content in this book in the public domain.

Have fun with it.

CHAPTER 5

Contributing

The project is hosted on GitHub and pull requests are welcome!