
# Flask Documentation

*Release 0.12.2*

Sep 27, 2017

# Contents

Part I

# USER'S GUIDE

This part of the documentation, which is mostly prose, begins with some background information about Flask, then focuses on step-by-step instructions for web development with Flask.

CHAPTER 1

# Foreword

Read this before you get started with Flask. This hopefully answers some questions about the purpose and goals of the project, and when you should or should not be using it.

## What does "micro" mean?

"Micro" does not mean that your whole web application has to fit into a single Python file (although it certainly can), nor does it mean that Flask is lacking in functionality. The "micro" in microframework means Flask aims to keep the core simple but extensible. Flask won't make many decisions for you, such as what database to use. Those decisions that it does make, such as what templating engine to use, are easy to change. Everything else is up to you, so that Flask can be everything you need and nothing you don't.

By default, Flask does not include a database abstraction layer, form validation or anything else where different libraries already exist that can handle that. Instead, Flask supports extensions to add such functionality to your application as if it was implemented in Flask itself. Numerous extensions provide database integration, form validation, upload handling, various open authentication technologies, and more. Flask may be "micro", but it's ready for production use on a variety of needs.

## Configuration and Conventions

Flask has many configuration values, with sensible defaults, and a few conventions when getting started. By convention, templates and static files are stored in subdi-

rectories within the application's Python source tree, with the names `templates` and `static` respectively. While this can be changed, you usually don't have to, especially when getting started.

## Growing with Flask

Once you have Flask up and running, you'll find a variety of extensions available in the community to integrate your project for production. The Flask core team reviews extensions and ensures approved extensions do not break with future releases.

As your codebase grows, you are free to make the design decisions appropriate for your project. Flask will continue to provide a very simple glue layer to the best that Python has to offer. You can implement advanced patterns in SQLAlchemy or another database tool, introduce non-relational data persistence as appropriate, and take advantage of framework-agnostic tools built for WSGI, the Python web interface.

Flask includes many hooks to customize its behavior. Should you need more customization, the Flask class is built for subclassing. If you are interested in that, check out the *Becoming Big* chapter. If you are curious about the Flask design principles, head over to the section about *Design Decisions in Flask*.

Continue to *Installation*, the *Quickstart*, or the *Foreword for Experienced Programmers*.

CHAPTER 2

# Foreword for Experienced Programmers

## Thread-Locals in Flask

One of the design decisions in Flask was that simple tasks should be simple; they should not take a lot of code and yet they should not limit you. Because of that, Flask has a few design choices that some people might find surprising or unorthodox. For example, Flask uses thread-local objects internally so that you don't have to pass objects around from function to function within a request in order to stay threadsafe. This approach is convenient, but requires a valid request context for dependency injection or when attempting to reuse code which uses a value pegged to the request. The Flask project is honest about thread-locals, does not hide them, and calls out in the code and documentation where they are used.

## Develop for the Web with Caution

Always keep security in mind when building web applications.

If you write a web application, you are probably allowing users to register and leave their data on your server. The users are entrusting you with data. And even if you are the only user that might leave data in your application, you still want that data to be stored securely.

Unfortunately, there are many ways the security of a web application can be compromised. Flask protects you against one of the most common security problems of modern web applications: cross-site scripting (XSS). Unless you deliberately mark insecure HTML as secure, Flask and the underlying Jinja2 template engine have you covered. But there are many more ways to cause security problems.

The documentation will warn you about aspects of web development that require attention to security. Some of these security concerns are far more complex than one might think, and we all sometimes underestimate the likelihood that a vulnerability will be exploited - until a clever attacker figures out a way to exploit our applications. And don't think that your application is not important enough to attract an attacker. Depending on the kind of attack, chances are that automated bots are probing for ways to fill your database with spam, links to malicious software, and the like.

Flask is no different from any other framework in that you the developer must build with caution, watching for exploits when building to your requirements.

## Python 3 Support in Flask

Flask, its dependencies, and most Flask extensions all support Python 3. If you want to use Flask with Python 3 have a look at the *Python 3 Support* page.

Continue to *Installation* or the *Quickstart*.

CHAPTER 3

# Installation

Flask depends on some external libraries, like Werkzeug and Jinja2. Werkzeug is a toolkit for WSGI, the standard Python interface between web applications and a variety of servers for both development and deployment. Jinja2 renders templates.

So how do you get all that on your computer quickly? There are many ways you could do that, but the most kick-ass method is virtualenv, so let's have a look at that first.

You will need Python 2.6 or newer to get started, so be sure to have an up-to-date Python 2.x installation. For using Flask with Python 3 have a look at *Python 3 Support*.

## virtualenv

Virtualenv is probably what you want to use during development, and if you have shell access to your production machines, you'll probably want to use it there, too.

What problem does virtualenv solve? If you like Python as much as I do, chances are you want to use it for other projects besides Flask-based web applications. But the more projects you have, the more likely it is that you will be working with different versions of Python itself, or at least different versions of Python libraries. Let's face it: quite often libraries break backwards compatibility, and it's unlikely that any serious application will have zero dependencies. So what do you do if two or more of your projects have conflicting dependencies?

Virtualenv to the rescue! Virtualenv enables multiple side-by-side installations of Python, one for each project. It doesn't actually install separate copies of Python, but it does provide a clever way to keep different project environments isolated. Let's see how virtualenv works.

If you are on Mac OS X or Linux, chances are that the following command will work for you:

```
$ sudo pip install virtualenv
```

It will probably install virtualenv on your system. Maybe it's even in your package manager. If you use Ubuntu, try:

```
$ sudo apt-get install python-virtualenv
```

If you are on Windows and don't have the `easy_install` command, you must install it first. Check the *pip and setuptools on Windows* section for more information about how to do that. Once you have it installed, run the same commands as above, but without the `sudo` prefix.

Once you have virtualenv installed, just fire up a shell and create your own environment. I usually create a project folder and a `venv` folder within:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip.............done.
```

Now, whenever you want to work on a project, you only have to activate the corresponding environment. On OS X and Linux, do the following:

```
$ . venv/bin/activate
```

If you are a Windows user, the following command is for you:

```
$ venv\Scripts\activate
```

Either way, you should now be using your virtualenv (notice how the prompt of your shell has changed to show the active environment).

And if you want to go back to the real world, use the following command:

```
$ deactivate
```

After doing this, the prompt of your shell should be as familiar as before.

Now, let's move on. Enter the following command to get Flask activated in your virtualenv:

```
$ pip install Flask
```

A few seconds later and you are good to go.

## System-Wide Installation

This is possible as well, though I do not recommend it. Just run `pip` with root privileges:

```
$ sudo pip install Flask
```

(On Windows systems, run it in a command-prompt window with administrator privileges, and leave out `sudo`.)

## Living on the Edge

If you want to work with the latest version of Flask, there are two ways: you can either let `pip` pull in the development version, or you can tell it to operate on a git checkout. Either way, virtualenv is recommended.

Get the git checkout in a new virtualenv and run in development mode:

```
$ git clone http://github.com/pallets/flask.git
Initialized empty Git repository in ~/dev/flask/.git/
$ cd flask
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip.............done.
$ . venv/bin/activate
$ python setup.py develop
...
Finished processing dependencies for Flask
```

This will pull in the dependencies and activate the git head as the current version inside the virtualenv. Then all you have to do is run `git pull origin` to update to the latest version.

## *pip* and *setuptools* on Windows

Sometimes getting the standard "Python packaging tools" like `pip`, `setuptools` and `virtualenv` can be a little trickier, but nothing very hard. The crucial package you will need is pip - this will let you install anything else (like virtualenv). Fortunately there is a "bootstrap script" you can run to install.

If you don't currently have `pip`, then *get-pip.py* will install it for you.

get-pip.py

It should be double-clickable once you download it. If you already have `pip`, you can upgrade them by running:

```
> pip install --upgrade pip setuptools
```

Most often, once you pull up a command prompt you want to be able to type `pip` and `python` which will run those things, but this might not automatically happen on Windows, because it doesn't know where those executables are (give either a try!).

To fix this, you should be able to navigate to your Python install directory (e.g `C:Python27`), then go to `Tools`, then `Scripts`, then find the `win_add2path.py` file and run that. Open a **new** Command Prompt and check that you can now just type `python` to bring up the interpreter.

Finally, to install *virtualenv*, you can simply run:

```
> pip install virtualenv
```

Then you can be off on your way following the installation instructions above.

CHAPTER 4

# Quickstart

Eager to get started? This page gives a good introduction to Flask. It assumes you already have Flask installed. If you do not, head over to the *Installation* section.

## A Minimal Application

A minimal Flask application looks something like this:

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

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

So what did that code do?

1. First we imported the `Flask` class. An instance of this class will be our WSGI application.
2. Next we create an instance of this class. The first argument is the name of the application's module or package. If you are using a single module (as in this example), you should use `__name__` because depending on if it's started as application or imported as module the name will be different (`'__main__'` versus the actual import name). This is needed so that Flask knows where to look for templates, static files, and so on. For more information have a look at the `Flask` documentation.

3. We then use the `route()` decorator to tell Flask what URL should trigger our function.
4. The function is given a name which is also used to generate URLs for that particular function, and returns the message we want to display in the user's browser.

Just save it as `hello.py` or something similar. Make sure to not call your application `flask.py` because this would conflict with Flask itself.

To run the application you can either use the **`flask`** command or python's `-m` switch with Flask. Before you can do that you need to tell your terminal the application to work with by exporting the `FLASK_APP` environment variable:

```
$ export FLASK_APP=hello.py
$ flask run
 * Running on http://127.0.0.1:5000/
```

If you are on Windows you need to use `set` instead of `export`.

Alternatively you can use **`python -m flask`**:

```
$ export FLASK_APP=hello.py
$ python -m flask run
 * Running on http://127.0.0.1:5000/
```

This launches a very simple builtin server, which is good enough for testing but probably not what you want to use in production. For deployment options see *Deployment Options*.

Now head over to http://127.0.0.1:5000/, and you should see your hello world greeting.

---

**Externally Visible Server**

If you run the server you will notice that the server is only accessible from your own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary Python code on your computer.

If you have the debugger disabled or trust the users on your network, you can make the server publicly available simply by adding `--host=0.0.0.0` to the command line:

```
flask run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

---

## What to do if the Server does not Start

In case the **`python -m flask`** fails or **`flask`** does not exist, there are multiple reasons this might be the case. First of all you need to look at the error message.

### Old Version of Flask

Versions of Flask older than 0.11 use to have different ways to start the application. In short, the **`flask`** command did not exist, and neither did **`python -m flask`**. In that case you have two options: either upgrade to newer Flask versions or have a look at the *Development Server* docs to see the alternative method for running a server.

### Invalid Import Name

The `FLASK_APP` environment variable is the name of the module to import at **`flask run`**. In case that module is incorrectly named you will get an import error upon start (or if debug is enabled when you navigate to the application). It will tell you what it tried to import and why it failed.

The most common reason is a typo or because you did not actually create an `app` object.

## Debug Mode

(Want to just log errors and stack traces? See *Application Errors*)

The **`flask`** script is nice to start a local development server, but you would have to restart it manually after each change to your code. That is not very nice and Flask can do better. If you enable debug support the server will reload itself on code changes, and it will also provide you with a helpful debugger if things go wrong.

To enable debug mode you can export the `FLASK_DEBUG` environment variable before running the server:

```
$ export FLASK_DEBUG=1
$ flask run
```

(On Windows you need to use `set` instead of `export`).

This does the following things:

1. it activates the debugger
2. it activates the automatic reloader
3. it enables the debug mode on the Flask application.

There are more parameters that are explained in the *Development Server* docs.

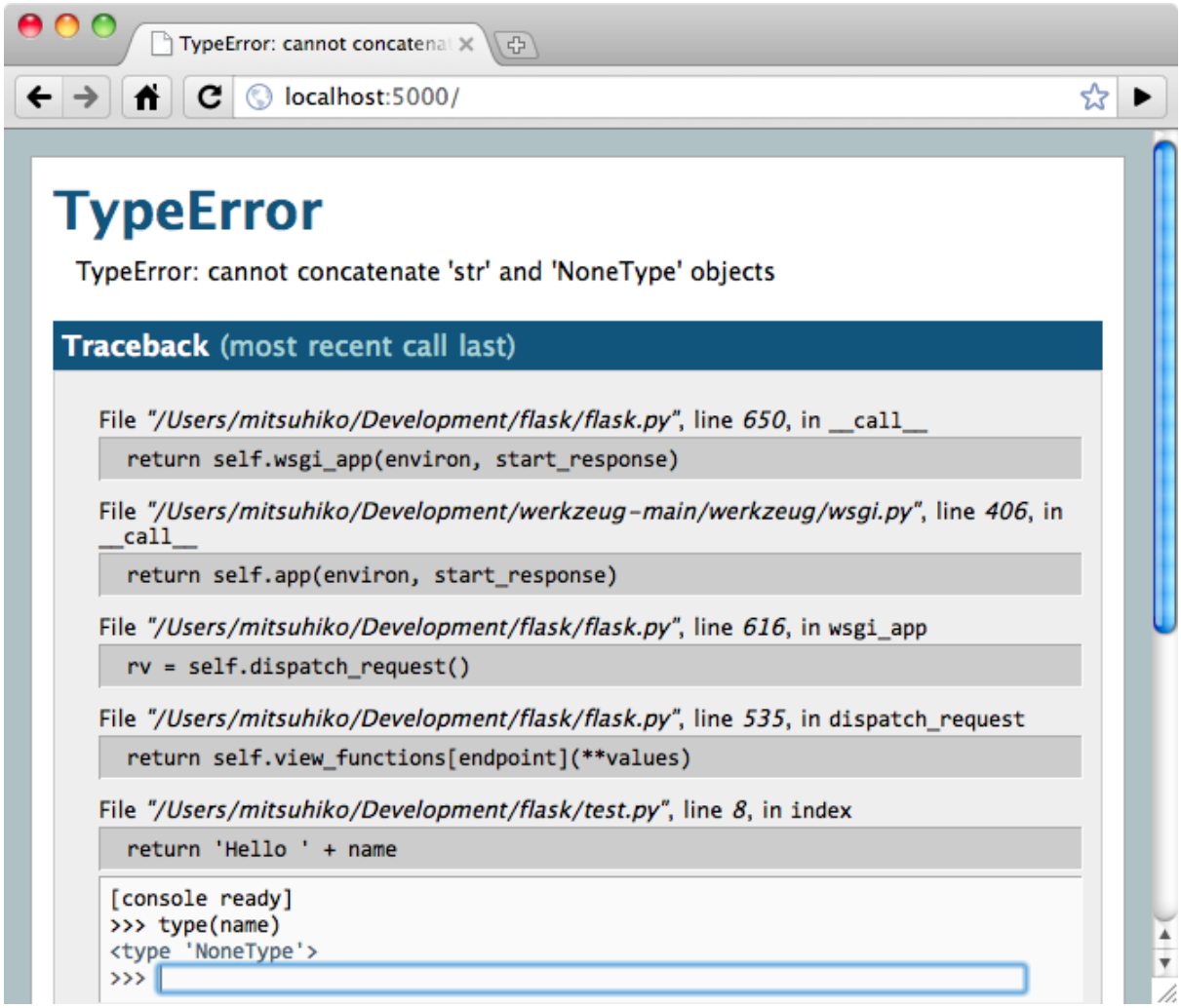
---

**Attention**

Even though the interactive debugger does not work in forking environments (which makes it nearly impossible to use on production servers), it still allows the execution of arbitrary code. This makes it a major security risk and therefore it **must never be used on production machines**.

---

Screenshot of the debugger in action:




Have another debugger in mind? See *Working with Debuggers*.

# Routing

Modern web applications have beautiful URLs. This helps people remember the URLs, which is especially handy for applications that are used from mobile devices with slower network connections. If the user can directly go to the desired page without having to hit the index page it is more likely they will like the page and come back next time.

As you have seen above, the `route()` decorator is used to bind a function to a URL. Here are some basic examples:

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
```

```
def hello():
    return 'Hello, World'
```

But there is more to it! You can make certain parts of the URL dynamic and attach multiple rules to a function.

## Variable Rules

To add variable parts to a URL you can mark these special sections as `<variable_name>`. Such a part is then passed as a keyword argument to your function. Optionally a converter can be used by specifying a rule with `<converter:variable_name>`. Here are some nice examples:

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id
```

The following converters exist:

| | |
|---|---|
| *string* | accepts any text without a slash (the default) |
| *int* | accepts integers |
| *float* | like `int` but for floating point values |
| *path* | like the default but also accepts slashes |
| *any* | matches one of the items provided |
| *uuid* | accepts UUID strings |

---

**Unique URLs / Redirection Behavior**

Flask's URL rules are based on Werkzeug's routing module. The idea behind that module is to ensure beautiful and unique URLs based on precedents laid down by Apache and earlier HTTP servers.

Take these two rules:

```
@app.route('/projects/')
def projects():
    return 'The project page'

@app.route('/about')
def about():
    return 'The about page'
```

Though they look rather similar, they differ in their use of the trailing slash in the URL *definition*. In the first case, the canonical URL for the `projects` endpoint has a trailing

slash. In that sense, it is similar to a folder on a filesystem. Accessing it without a trailing slash will cause Flask to redirect to the canonical URL with the trailing slash.

In the second case, however, the URL is defined without a trailing slash, rather like the pathname of a file on UNIX-like systems. Accessing the URL with a trailing slash will produce a 404 "Not Found" error.

This behavior allows relative URLs to continue working even if the trailing slash is omitted, consistent with how Apache and other servers work. Also, the URLs will stay unique, which helps search engines avoid indexing the same page twice.

---

## URL Building

If it can match URLs, can Flask also generate them? Of course it can. To build a URL to a specific function you can use the `url_for()` function. It accepts the name of the function as first argument and a number of keyword arguments, each corresponding to the variable part of the URL rule. Unknown variable parts are appended to the URL as query parameters. Here are some examples:

```
>>> from flask import Flask, url_for
>>> app = Flask(__name__)
>>> @app.route('/')
... def index(): pass
...
>>> @app.route('/login')
... def login(): pass
...
>>> @app.route('/user/<username>')
... def profile(username): pass
...
>>> with app.test_request_context():
...  print url_for('index')
...  print url_for('login')
...  print url_for('login', next='/')
...  print url_for('profile', username='John Doe')
...
/
/login
/login?next=/
/user/John%20Doe
```

(This also uses the `test_request_context()` method, explained below. It tells Flask to behave as though it is handling a request, even though we are interacting with it through a Python shell. Have a look at the explanation below. *Context Locals*).

Why would you want to build URLs using the URL reversing function `url_for()` instead of hard-coding them into your templates? There are three good reasons for this:

1. Reversing is often more descriptive than hard-coding the URLs. More importantly, it allows you to change URLs in one go, without having to remember to

change URLs all over the place.

2. URL building will handle escaping of special characters and Unicode data transparently for you, so you don't have to deal with them.
3. If your application is placed outside the URL root - say, in `/myapplication` instead of `/` - `url_for()` will handle that properly for you.

## HTTP Methods

HTTP (the protocol web applications are speaking) knows different methods for accessing URLs. By default, a route only answers to `GET` requests, but that can be changed by providing the `methods` argument to the `route()` decorator. Here are some examples:

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

If `GET` is present, `HEAD` will be added automatically for you. You don't have to deal with that. It will also make sure that `HEAD` requests are handled as the HTTP RFC (the document describing the HTTP protocol) demands, so you can completely ignore that part of the HTTP specification. Likewise, as of Flask 0.6, `OPTIONS` is implemented for you automatically as well.

You have no idea what an HTTP method is? Worry not, here is a quick introduction to HTTP methods and why they matter:

The HTTP method (also often called "the verb") tells the server what the client wants to *do* with the requested page. The following methods are very common:

**`GET`** The browser tells the server to just *get* the information stored on that page and send it. This is probably the most common method.

**`HEAD`** The browser tells the server to get the information, but it is only interested in the *headers*, not the content of the page. An application is supposed to handle that as if a `GET` request was received but to not deliver the actual content. In Flask you don't have to deal with that at all, the underlying Werkzeug library handles that for you.

**`POST`** The browser tells the server that it wants to *post* some new information to that URL and that the server must ensure the data is stored and only stored once. This is how HTML forms usually transmit data to the server.

**`PUT`** Similar to `POST` but the server might trigger the store procedure multiple times by overwriting the old values more than once. Now you might be asking why this is useful, but there are some good reasons to do it this way. Consider that the connection is lost during transmission: in this situation a system between the

browser and the server might receive the request safely a second time without breaking things. With `POST` that would not be possible because it must only be triggered once.

**`DELETE`** Remove the information at the given location.

**`OPTIONS`** Provides a quick way for a client to figure out which methods are supported by this URL. Starting with Flask 0.6, this is implemented for you automatically.

Now the interesting part is that in HTML4 and XHTML1, the only methods a form can submit to the server are `GET` and `POST`. But with JavaScript and future HTML standards you can use the other methods as well. Furthermore HTTP has become quite popular lately and browsers are no longer the only clients that are using HTTP. For instance, many revision control systems use it.

## Static Files

Dynamic web applications also need static files. That's usually where the CSS and JavaScript files are coming from. Ideally your web server is configured to serve them for you, but during development Flask can do that as well. Just create a folder called `static` in your package or next to your module and it will be available at `/static` on the application.

To generate URLs for static files, use the special `'static'` endpoint name:

```
url_for('static', filename='style.css')
```

The file has to be stored on the filesystem as `static/style.css`.

## Rendering Templates

Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure. Because of that Flask configures the Jinja2 template engine for you automatically.

To render a template you can use the `render_template()` method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here's a simple example of how to render a template:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask will look for templates in the `templates` folder. So if your application is a module, this folder is next to that module, if it's a package it's actually inside your package:

**Case 1**: a module:

```
/application.py
/templates
    /hello.html
```

**Case 2**: a package:

```
/application
    /__init__.py
    /templates
        /hello.html
```

For templates you can use the full power of Jinja2 templates. Head over to the official Jinja2 Template Documentation for more information.

Here is an example template:

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

Inside templates you also have access to the `request`, `session` and `g`[1] objects as well as the `get_flashed_messages()` function.

Templates are especially useful if inheritance is used. If you want to know how that works, head over to the *Template Inheritance* pattern documentation. Basically template inheritance makes it possible to keep certain elements on each page (like header, navigation and footer).

Automatic escaping is enabled, so if `name` contains HTML it will be escaped automatically. If you can trust a variable and you know that it will be safe HTML (for example because it came from a module that converts wiki markup to HTML) you can mark it as safe by using the `Markup` class or by using the `|safe` filter in the template. Head over to the Jinja 2 documentation for more examples.

Here is a basic introduction to how the `Markup` class works:

```
>>> from flask import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
Markup(u'<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
>>> Markup.escape('<blink>hacker</blink>')
Markup(u'&lt;blink&gt;hacker&lt;/blink&gt;')
>>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
u'Marked up \xbb HTML'
```

[1] Unsure what that g object is? It's something in which you can store information for your own needs, check the documentation of that object (g) and the *Using SQLite 3 with Flask* for more information.

Changed in version 0.5: Autoescaping is no longer enabled for all templates. The following extensions for templates trigger autoescaping: `.html`, `.htm`, `.xml`, `.xhtml`. Templates loaded from a string will have autoescaping disabled.

# Accessing Request Data

For web applications it's crucial to react to the data a client sends to the server. In Flask this information is provided by the global `request` object. If you have some experience with Python you might be wondering how that object can be global and how Flask manages to still be threadsafe. The answer is context locals:

## Context Locals

---

**Insider Information**

If you want to understand how that works and how you can implement tests with context locals, read this section, otherwise just skip it.

---

Certain objects in Flask are global objects, but not of the usual kind. These objects are actually proxies to objects that are local to a specific context. What a mouthful. But that is actually quite easy to understand.

Imagine the context being the handling thread. A request comes in and the web server decides to spawn a new thread (or something else, the underlying object is capable of dealing with concurrency systems other than threads). When Flask starts its internal request handling it figures out that the current thread is the active context and binds the current application and the WSGI environments to that context (thread). It does that in an intelligent way so that one application can invoke another application without breaking.

So what does this mean to you? Basically you can completely ignore that this is the case unless you are doing something like unit testing. You will notice that code which depends on a request object will suddenly break because there is no request object. The solution is creating a request object yourself and binding it to the context. The easiest solution for unit testing is to use the `test_request_context()` context manager. In combination with the `with` statement it will bind a test request so that you can interact with it. Here is an example:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'
```

The other possibility is passing a whole WSGI environment to the `request_context()` method:

```
from flask import request

with app.request_context(environ):
    assert request.method == 'POST'
```

## The Request Object

The request object is documented in the API section and we will not cover it here in detail (see `request`). Here is a broad overview of some of the most common operations. First of all you have to import it from the `flask` module:

```
from flask import request
```

The current request method is available by using the `method` attribute. To access form data (data transmitted in a `POST` or `PUT` request) you can use the `form` attribute. Here is a full example of the two attributes mentioned above:

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return render_template('login.html', error=error)
```

What happens if the key does not exist in the `form` attribute? In that case a special `KeyError` is raised. You can catch it like a standard `KeyError` but if you don't do that, a HTTP 400 Bad Request error page is shown instead. So for many situations you don't have to deal with that problem.

To access parameters submitted in the URL (`?key=value`) you can use the `args` attribute:

```
searchword = request.args.get('key', '')
```

We recommend accessing URL parameters with *get* or by catching the `KeyError` because users might change the URL and presenting them a 400 bad request page in that case is not user friendly.

For a full list of methods and attributes of the request object, head over to the `request` documentation.

## File Uploads

You can handle uploaded files with Flask easily. Just make sure not to forget to set the `enctype="multipart/form-data"` attribute on your HTML form, otherwise the browser will not transmit your files at all.

Uploaded files are stored in memory or at a temporary location on the filesystem. You can access those files by looking at the `files` attribute on the request object. Each uploaded file is stored in that dictionary. It behaves just like a standard Python `file` object, but it also has a `save()` method that allows you to store that file on the filesystem of the server. Here is a simple example showing how that works:

```
from flask import request

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/uploaded_file.txt')
    ...
```

If you want to know how the file was named on the client before it was uploaded to your application, you can access the `filename` attribute. However please keep in mind that this value can be forged so never ever trust that value. If you want to use the filename of the client to store the file on the server, pass it through the `secure_filename()` function that Werkzeug provides for you:

```
from flask import request
from werkzeug.utils import secure_filename

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/' + secure_filename(f.filename))
    ...
```

For some better examples, checkout the *Uploading Files* pattern.

## Cookies

To access cookies you can use the `cookies` attribute. To set cookies you can use the `set_cookie` method of response objects. The `cookies` attribute of request objects is a dictionary with all the cookies the client transmits. If you want to use sessions, do not use the cookies directly but instead use the *Sessions* in Flask that add some security on top of cookies for you.

Reading cookies:

```
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    # use cookies.get(key) instead of cookies[key] to not get a
    # KeyError if the cookie is missing.
```

Storing cookies:

```
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

Note that cookies are set on response objects. Since you normally just return strings from the view functions Flask will convert them into response objects for you. If you explicitly want to do that you can use the `make_response()` function and then modify it.

Sometimes you might want to set a cookie at a point where the response object does not exist yet. This is possible by utilizing the *Deferred Request Callbacks* pattern.

For this also see *About Responses*.

## Redirects and Errors

To redirect a user to another endpoint, use the `redirect()` function; to abort a request early with an error code, use the `abort()` function:

```
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

This is a rather pointless example because a user will be redirected from the index to a page they cannot access (401 means access denied) but it shows how that works.

By default a black and white error page is shown for each error code. If you want to customize the error page, you can use the `errorhandler()` decorator:

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(error):
    return render_template('page_not_found.html'), 404
```

Note the `404` after the `render_template()` call. This tells Flask that the status code of that page should be 404 which means not found. By default 200 is assumed which translates to: all went well.

See *Error handlers* for more details.

## About Responses

The return value from a view function is automatically converted into a response object for you. If the return value is a string it's converted into a response object with the string as response body, a `200 OK` status code and a `text/html` mimetype. The logic that Flask applies to converting return values into response objects is as follows:

1. If a response object of the correct type is returned it's directly returned from the view.
2. If it's a string, a response object is created with that data and the default parameters.
3. If a tuple is returned the items in the tuple can provide extra information. Such tuples have to be in the form `(response, status, headers)` or `(response, headers)` where at least one item has to be in the tuple. The `status` value will override the status code and `headers` can be a list or dictionary of additional header values.
4. If none of that works, Flask will assume the return value is a valid WSGI application and convert that into a response object.

If you want to get hold of the resulting response object inside the view you can use the `make_response()` function.

Imagine you have a view like this:

```
@app.errorhandler(404)
def not_found(error):
    return render_template('error.html'), 404
```

You just need to wrap the return expression with `make_response()` and get the response object to modify it, then return it:

```
@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
```

```
    resp.headers['X-Something'] = 'A value'
    return resp
```

## Sessions

In addition to the request object there is also a second object called `session` which allows you to store information specific to a user from one request to the next. This is implemented on top of cookies for you and signs the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless they know the secret key used for signing.

In order to use sessions you have to set a secret key. Here is how sessions work:

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

app = Flask(__name__)

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

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))

# set the secret key.  keep this really secret:
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
```

The `escape()` mentioned here does escaping for you if you are not using the template engine (as in this example).

**How to generate good secret keys**

The problem with random is that it's hard to judge what is truly random. And a secret key should be as random as possible. Your operating system has ways to generate pretty random stuff based on a cryptographic random generator which can be used to get such a key:

```
>>> import os
>>> os.urandom(24)
'\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x01O<!\xd5\xa2\xa0\x9fR"\xa1\xa8'

Just take that thing and copy/paste it into your code and you're done.
```

A note on cookie-based sessions: Flask will take the values you put into the session object and serialize them into a cookie. If you are finding some values do not persist across requests, cookies are indeed enabled, and you are not getting a clear error message, check the size of the cookie in your page responses compared to the size supported by web browsers.

Besides the default client-side based sessions, if you want to handle sessions on the server-side instead, there are several Flask extensions that support this.

## Message Flashing

Good applications and user interfaces are all about feedback. If the user does not get enough feedback they will probably end up hating the application. Flask provides a really simple way to give feedback to a user with the flashing system. The flashing system basically makes it possible to record a message at the end of a request and access it on the next (and only the next) request. This is usually combined with a layout template to expose the message.

To flash a message use the `flash()` method, to get hold of the messages you can use `get_flashed_messages()` which is also available in the templates. Check out the *Message Flashing* for a full example.

## Logging

New in version 0.3.

Sometimes you might be in a situation where you deal with data that should be correct, but actually is not. For example you may have some client-side code that sends an HTTP request to the server but it's obviously malformed. This might be caused by a user tampering with the data, or the client code failing. Most of the time it's okay to reply with `400 Bad Request` in that situation, but sometimes that won't do and the code has to continue working.

You may still want to log that something fishy happened. This is where loggers come in handy. As of Flask 0.3 a logger is preconfigured for you to use.

Here are some example log calls:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

The attached `logger` is a standard logging `Logger`, so head over to the official logging documentation for more information.

Read more on *Application Errors*.

## Hooking in WSGI Middlewares

If you want to add a WSGI middleware to your application you can wrap the internal WSGI application. For example if you want to use one of the middlewares from the Werkzeug package to work around bugs in lighttpd, you can do it like this:

```
from werkzeug.contrib.fixers import LighttpdCGIRootFix
app.wsgi_app = LighttpdCGIRootFix(app.wsgi_app)
```

## Using Flask Extensions

Extensions are packages that help you accomplish common tasks. For example, Flask-SQLAlchemy provides SQLAlchemy support that makes it simple and easy to use with Flask.

For more on Flask extensions, have a look at *Flask Extensions*.

## Deploying to a Web Server

Ready to deploy your new Flask app? Go to *Deployment Options*.

CHAPTER 5

# Tutorial

You want to develop an application with Python and Flask? Here you have the chance to learn by example. In this tutorial, we will create a simple microblogging application. It only supports one user that can create text-only entries and there are no feeds or comments, but it still features everything you need to get started. We will use Flask and SQLite as a database (which comes out of the box with Python) so there is nothing else you need.

If you want the full source code in advance or for comparison, check out the example source.

## Introducing Flaskr

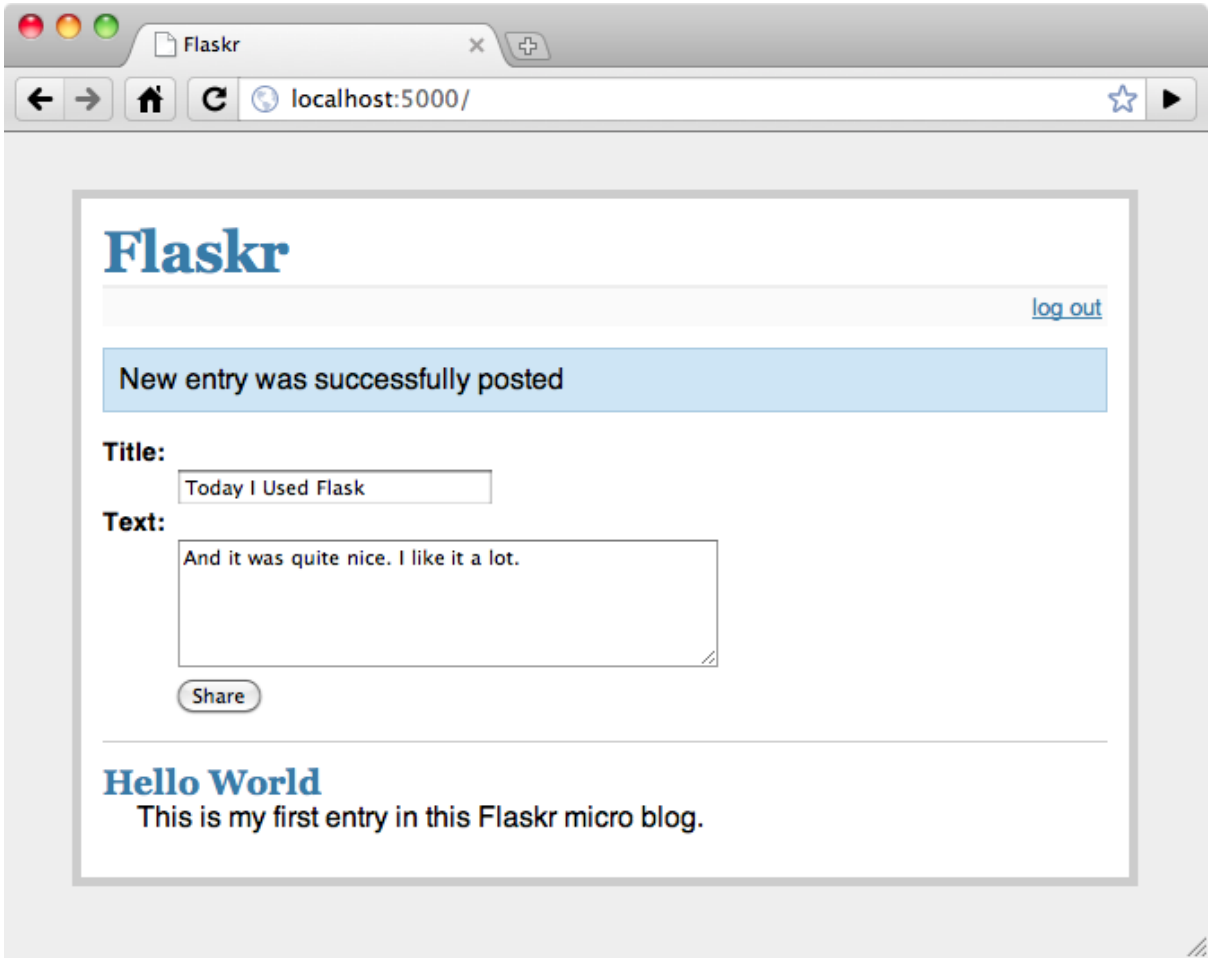
This tutorial will demonstrate a blogging application named Flaskr, but feel free to choose your own less Web-2.0-ish name ;) Essentially, it will do the following things:

1. Let the user sign in and out with credentials specified in the configuration. Only one user is supported.
2. When the user is logged in, they can add new entries to the page consisting of a text-only title and some HTML for the text. This HTML is not sanitized because we trust the user here.
3. The index page shows all entries so far in reverse chronological order (newest on top) and the user can add new ones from there if logged in.

SQLite3 will be used directly for this application because it's good enough for an application of this size. For larger applications, however, it makes a lot of sense to use SQLAlchemy, as it handles database connections in a more intelligent way, allowing

you to target different relational databases at once and more. You might also want to consider one of the popular NoSQL databases if your data is more suited for those.

Here a screenshot of the final application:




Continue with *Step 0: Creating The Folders*.

## Step 0: Creating The Folders

Before getting started, you will need to create the folders needed for this application:

```
/flaskr
    /flaskr
        /static
        /templates
```

The application will be installed and run as Python package. This is the recommended way to install and run Flask applications. You will see exactly how to run `flaskr` later on in this tutorial. For now go ahead and create the applications directory structure. In the next few steps you will be creating the database schema as well as the main module.

As a quick side note, the files inside of the `static` folder are available to users of the

application via HTTP. This is the place where CSS and JavaScript files go. Inside the `templates` folder, Flask will look for Jinja2 templates. You will see examples of this later on.

For now you should continue with *Step 1: Database Schema*.

## Step 1: Database Schema

In this step, you will create the database schema. Only a single table is needed for this application and it will only support SQLite. All you need to do is put the following contents into a file named `schema.sql` in the `flaskr/flaskr` folder:

```
drop table if exists entries;
create table entries (
  id integer primary key autoincrement,
  title text not null,
  'text' text not null
);
```

This schema consists of a single table called `entries`. Each row in this table has an `id`, a `title`, and a `text`. The `id` is an automatically incrementing integer and a primary key, the other two are strings that must not be null.

Continue with *Step 2: Application Setup Code*.

## Step 2: Application Setup Code

Now that the schema is in place, you can create the application module, `flaskr.py`. This file should be placed inside of the `flaskr/flaskr` folder. The first several lines of code in the application module are the needed import statements. After that there will be a few lines of configuration code. For small applications like `flaskr`, it is possible to drop the configuration directly into the module. However, a cleaner solution is to create a separate `.ini` or `.py` file, load that, and import the values from there.

Here are the import statements (in `flaskr.py`):

```
# all the imports
import os
import sqlite3
from flask import Flask, request, session, g, redirect, url_for, abort, \
     render_template, flash
```

The next couple lines will create the actual application instance and initialize it with the config from the same file in `flaskr.py`:

```
app = Flask(__name__) # create the application instance :)
app.config.from_object(__name__) # load config from this file , flaskr.py
```

```
# Load default config and override config from an environment variable
app.config.update(dict(
    DATABASE=os.path.join(app.root_path, 'flaskr.db'),
    SECRET_KEY='development key',
    USERNAME='admin',
    PASSWORD='default'
))
app.config.from_envvar('FLASKR_SETTINGS', silent=True)
```

The `Config` object works similarly to a dictionary, so it can be updated with new values.

---

**Database Path**

Operating systems know the concept of a current working directory for each process. Unfortunately, you cannot depend on this in web applications because you might have more than one application in the same process.

For this reason the `app.root_path` attribute can be used to get the path to the application. Together with the `os.path` module, files can then easily be found. In this example, we place the database right next to it.

For a real-world application, it's recommended to use *Instance Folders* instead.

---

Usually, it is a good idea to load a separate, environment-specific configuration file. Flask allows you to import multiple configurations and it will use the setting defined in the last import. This enables robust configuration setups. `from_envvar()` can help achieve this.

```
app.config.from_envvar('FLASKR_SETTINGS', silent=True)
```

Simply define the environment variable `FLASKR_SETTINGS` that points to a config file to be loaded. The silent switch just tells Flask to not complain if no such environment key is set.

In addition to that, you can use the `from_object()` method on the config object and provide it with an import name of a module. Flask will then initialize the variable from that module. Note that in all cases, only variable names that are uppercase are considered.

The `SECRET_KEY` is needed to keep the client-side sessions secure. Choose that key wisely and as hard to guess and complex as possible.

Lastly, you will add a method that allows for easy connections to the specified database. This can be used to open a connection on request and also from the interactive Python shell or a script. This will come in handy later. You can create a simple database connection through SQLite and then tell it to use the `sqlite3.Row` object to represent rows. This allows the rows to be treated as if they were dictionaries instead of tuples.

```
def connect_db():
    """Connects to the specific database."""
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv
```

In the next section you will see how to run the application.

Continue with *Step 3: Installing flaskr as a Package*.

## Step 3: Installing flaskr as a Package

Flask is now shipped with built-in support for Click. Click provides Flask with enhanced and extensible command line utilities. Later in this tutorial you will see exactly how to extend the `flask` command line interface (CLI).

A useful pattern to manage a Flask application is to install your app following the Python Packaging Guide. Presently this involves creating two new files; `setup.py` and `MANIFEST.in` in the projects root directory. You also need to add an `__init__.py` file to make the `flaskr/flaskr` directory a package. After these changes, your code structure should be:

```
/flaskr
    /flaskr
        __init__.py
        /static
        /templates
        flaskr.py
        schema.sql
    setup.py
    MANIFEST.in
```

The content of the `setup.py` file for `flaskr` is:

```
from setuptools import setup

setup(
    name='flaskr',
    packages=['flaskr'],
    include_package_data=True,
    install_requires=[
        'flask',
    ],
)
```

When using setuptools, it is also necessary to specify any special files that should be included in your package (in the `MANIFEST.in`). In this case, the static and templates directories need to be included, as well as the schema. Create the `MANIFEST.in` and add the following lines:

```
graft flaskr/templates
graft flaskr/static
include flaskr/schema.sql
```

To simplify locating the application, add the following import statement into this file, `flaskr/__init__.py`:

```
from .flaskr import app
```

This import statement brings the application instance into the top-level of the application package. When it is time to run the application, the Flask development server needs the location of the app instance. This import statement simplifies the location process. Without it the export statement a few steps below would need to be `export FLASK_APP=flaskr.flaskr`.

At this point you should be able to install the application. As usual, it is recommended to install your Flask application within a virtualenv. With that said, go ahead and install the application with:

```
pip install --editable .
```

The above installation command assumes that it is run within the projects root directory, *flaskr/*. The *editable* flag allows editing source code without having to reinstall the Flask app each time you make changes. The flaskr app is now installed in your virtualenv (see output of `pip freeze`).

With that out of the way, you should be able to start up the application. Do this with the following commands:

```
export FLASK_APP=flaskr
export FLASK_DEBUG=true
flask run
```

(In case you are on Windows you need to use *set* instead of *export*). The `FLASK_DEBUG` flag enables or disables the interactive debugger. *Never leave debug mode activated in a production system*, because it will allow users to execute code on the server!

You will see a message telling you that server has started along with the address at which you can access it.

When you head over to the server in your browser, you will get a 404 error because we don't have any views yet. That will be addressed a little later, but first, you should get the database working.

---

**Externally Visible Server**

Want your server to be publicly available? Check out the *externally visible server* section for more information.

---

Continue with *Step 4: Database Connections*.

## Step 4: Database Connections

You currently have a function for establishing a database connection with *connect_db*, but by itself, it is not particularly useful. Creating and closing database connections all the time is very inefficient, so you will need to keep it around for longer. Because database connections encapsulate a transaction, you will need to make sure that only one request at a time uses the connection. An elegant way to do this is by utilizing the *application context*.

Flask provides two contexts: the *application context* and the *request context*. For the time being, all you have to know is that there are special variables that use these. For instance, the `request` variable is the request object associated with the current request, whereas `g` is a general purpose variable associated with the current application context. The tutorial will cover some more details of this later on.

For the time being, all you have to know is that you can store information safely on the `g` object.

So when do you put it on there? To do that you can make a helper function. The first time the function is called, it will create a database connection for the current context, and successive calls will return the already established connection:

```
def get_db():
    """Opens a new database connection if there is none yet for the
    current application context.
    """
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

Now you know how to connect, but how can you properly disconnect? For that, Flask provides us with the `teardown_appcontext()` decorator. It's executed every time the application context tears down:

```
@app.teardown_appcontext
def close_db(error):
    """Closes the database again at the end of the request."""
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```

Functions marked with `teardown_appcontext()` are called every time the app context tears down. What does this mean? Essentially, the app context is created before the request comes in and is destroyed (torn down) whenever the request finishes. A teardown can happen because of two reasons: either everything went well (the error parameter will be `None`) or an exception happened, in which case the error is passed to the teardown function.

Curious about what these contexts mean? Have a look at the *The Application Context* documentation to learn more.

Continue to *Step 5: Creating The Database*.

**Hint:** Where do I put this code?

If you've been following along in this tutorial, you might be wondering where to put the code from this step and the next. A logical place is to group these module-level functions together, and put your new `get_db` and `close_db` functions below your existing `connect_db` function (following the tutorial line-by-line).

If you need a moment to find your bearings, take a look at how the example source is organized. In Flask, you can put all of your application code into a single Python module. You don't have to, and if your app *grows larger*, it's a good idea not to.

## Step 5: Creating The Database

As outlined earlier, Flaskr is a database powered application, and more precisely, it is an application powered by a relational database system. Such systems need a schema that tells them how to store that information. Before starting the server for the first time, it's important to create that schema.

Such a schema can be created by piping the `schema.sql` file into the *sqlite3* command as follows:

```
sqlite3 /tmp/flaskr.db < schema.sql
```

The downside of this is that it requires the `sqlite3` command to be installed, which is not necessarily the case on every system. This also requires that you provide the path to the database, which can introduce errors. It's a good idea to add a function that initializes the database for you, to the application.

To do this, you can create a function and hook it into a **`flask`** command that initializes the database. For now just take a look at the code segment below. A good place to add this function, and command, is just below the *connect_db* function in `flaskr.py`:

```
def init_db():
    db = get_db()
    with app.open_resource('schema.sql', mode='r') as f:
        db.cursor().executescript(f.read())
    db.commit()

@app.cli.command('initdb')
def initdb_command():
    """Initializes the database."""
    init_db()
    print('Initialized the database.')
```

The `app.cli.command()` decorator registers a new command with the **`flask`** script. When the command executes, Flask will automatically create an application context which is bound to the right application. Within the function, you can then access

`flask.g` and other things as you might expect. When the script ends, the application context tears down and the database connection is released.

You will want to keep an actual function around that initializes the database, though, so that we can easily create databases in unit tests later on. (For more information see *Testing Flask Applications*.)

The `open_resource()` method of the application object is a convenient helper function that will open a resource that the application provides. This function opens a file from the resource location (the `flaskr/flaskr` folder) and allows you to read from it. It is used in this example to execute a script on the database connection.

The connection object provided by SQLite can give you a cursor object. On that cursor, there is a method to execute a complete script. Finally, you only have to commit the changes. SQLite3 and other transactional databases will not commit unless you explicitly tell it to.

Now, it is possible to create a database with the **`flask`** script:

```
flask initdb
Initialized the database.
```

---

**Troubleshooting**

If you get an exception later on stating that a table cannot be found, check that you did execute the `initdb` command and that your table names are correct (singular vs. plural, for example).

---

Continue with *Step 6: The View Functions*

# Step 6: The View Functions

Now that the database connections are working, you can start writing the view functions. You will need four of them:

## Show Entries

This view shows all the entries stored in the database. It listens on the root of the application and will select title and text from the database. The one with the highest id (the newest entry) will be on top. The rows returned from the cursor look a bit like dictionaries because we are using the `sqlite3.Row` row factory.

The view function will pass the entries to the `show_entries.html` template and return the rendered one:

```
@app.route('/')
def show_entries():
```

```
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('show_entries.html', entries=entries)
```

## Add New Entry

This view lets the user add new entries if they are logged in. This only responds to `POST` requests; the actual form is shown on the *show_entries* page. If everything worked out well, it will `flash()` an information message to the next request and redirect back to the *show_entries* page:

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

Note that this view checks that the user is logged in (that is, if the *logged_in* key is present in the session and `True`).

---

**Security Note**

Be sure to use question marks when building SQL statements, as done in the example above. Otherwise, your app will be vulnerable to SQL injection when you use string formatting to build SQL statements. See *Using SQLite 3 with Flask* for more.

---

## Login and Logout

These functions are used to sign the user in and out. Login checks the username and password against the ones from the configuration and sets the *logged_in* key for the session. If the user logged in successfully, that key is set to `True`, and the user is redirected back to the *show_entries* page. In addition, a message is flashed that informs the user that he or she was logged in successfully. If an error occurred, the template is notified about that, and the user is asked again:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
```

```
            error = 'Invalid username'
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_entries'))
    return render_template('login.html', error=error)
```

The *logout* function, on the other hand, removes that key from the session again. There is a neat trick here: if you use the `pop()` method of the dict and pass a second parameter to it (the default), the method will delete the key from the dictionary if present or do nothing when that key is not in there. This is helpful because now it is not necessary to check if the user was logged in.

```
@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('show_entries'))
```

---

**Security Note**

Passwords should never be stored in plain text in a production system. This tutorial uses plain text passwords for simplicity. If you plan to release a project based off this tutorial out into the world, passwords should be both hashed and salted before being stored in a database or file.

Fortunately, there are Flask extensions for the purpose of hashing passwords and verifying passwords against hashes, so adding this functionality is fairly straight forward. There are also many general python libraries that can be used for hashing.

You can find a list of recommended Flask extensions here

---

Continue with *Step 7: The Templates*.

# Step 7: The Templates

Now it is time to start working on the templates. As you may have noticed, if you make requests with the app running, you will get an exception that Flask cannot find the templates. The templates are using Jinja2 syntax and have autoescaping enabled by default. This means that unless you mark a value in the code with `Markup` or with the `|safe` filter in the template, Jinja2 will ensure that special characters such as < or > are escaped with their XML equivalents.

We are also using template inheritance which makes it possible to reuse the layout of the website in all pages.

Put the following templates into the `templates` folder:

## layout.html

This template contains the HTML skeleton, the header and a link to log in (or log out if the user was already logged in). It also displays the flashed messages if there are any. The `{% block body %}` block can be replaced by a block of the same name (`body`) in a child template.

The `session` dict is available in the template as well and you can use that to check if the user is logged in or not. Note that in Jinja you can access missing attributes and items of objects / dicts which makes the following code work, even if there is no `'logged_in'` key in the session:

```
<!doctype html>
<title>Flaskr</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='style.css
↪') }}">
<div class=page>
  <h1>Flaskr</h1>
  <div class=metanav>
  {% if not session.logged_in %}
    <a href="{{ url_for('login') }}">log in</a>
  {% else %}
    <a href="{{ url_for('logout') }}">log out</a>
  {% endif %}
  </div>
  {% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
  {% block body %}{% endblock %}
</div>
```

## show_entries.html

This template extends the `layout.html` template from above to display the messages. Note that the `for` loop iterates over the messages we passed in with the `render_template()` function. Notice that the form is configured to to submit to the *add_entry* view function and use `POST` as HTTP method:

```
{% extends "layout.html" %}
{% block body %}
  {% if session.logged_in %}
    <form action="{{ url_for('add_entry') }}" method=post class=add-entry>
      <dl>
        <dt>Title:
        <dd><input type=text size=30 name=title>
        <dt>Text:
```

```
        <dd><textarea name=text rows=5 cols=40></textarea>
        <dd><input type=submit value=Share>
      </dl>
    </form>
  {% endif %}
  <ul class=entries>
  {% for entry in entries %}
    <li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}
  {% else %}
    <li><em>Unbelievable.  No entries here so far</em>
  {% endfor %}
  </ul>
{% endblock %}
```

### login.html

This is the login template, which basically just displays a form to allow the user to login:

```
{% extends "layout.html" %}
{% block body %}
  <h2>Login</h2>
  {% if error %}<p class=error><strong>Error:</strong> {{ error }}{% endif %}
  <form action="{{ url_for('login') }}" method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username>
      <dt>Password:
      <dd><input type=password name=password>
      <dd><input type=submit value=Login>
    </dl>
  </form>
{% endblock %}
```

Continue with *Step 8: Adding Style*.

## Step 8: Adding Style

Now that everything else works, it's time to add some style to the application. Just create a stylesheet called `style.css` in the `static` folder:

```
body            { font-family: sans-serif; background: #eee; }
a, h1, h2       { color: #377ba8; }
h1, h2          { font-family: 'Georgia', serif; margin: 0; }
h1              { border-bottom: 2px solid #eee; }
h2              { font-size: 1.2em; }
```

```
.page           { margin: 2em auto; width: 35em; border: 5px solid #ccc;
                  padding: 0.8em; background: white; }
.entries        { list-style: none; margin: 0; padding: 0; }
.entries li     { margin: 0.8em 1.2em; }
.entries li h2  { margin-left: -1em; }
.add-entry      { font-size: 0.9em; border-bottom: 1px solid #ccc; }
.add-entry dl   { font-weight: bold; }
.metanav        { text-align: right; font-size: 0.8em; padding: 0.3em;
                  margin-bottom: 1em; background: #fafafa; }
.flash          { background: #cee5F5; padding: 0.5em;
                  border: 1px solid #aacbe2; }
.error          { background: #f0d6d6; padding: 0.5em; }
```

Continue with *Bonus: Testing the Application*.

# Bonus: Testing the Application

Now that you have finished the application and everything works as expected, it's probably not a bad idea to add automated tests to simplify modifications in the future. The application above is used as a basic example of how to perform unit testing in the *Testing Flask Applications* section of the documentation. Go there to see how easy it is to test Flask applications.

## Adding tests to flaskr

Assuming you have seen the *Testing Flask Applications* section and have either written your own tests for `flaskr` or have followed along with the examples provided, you might be wondering about ways to organize the project.

One possible and recommended project structure is:

```
flaskr/
    flaskr/
        __init__.py
        static/
        templates/
    tests/
        test_flaskr.py
    setup.py
    MANIFEST.in
```

For now go ahead a create the `tests/` directory as well as the `test_flaskr.py` file.

## Running the tests

At this point you can run the tests. Here `pytest` will be used.

**Note:** Make sure that `pytest` is installed in the same virtualenv as flaskr. Otherwise `pytest` test will not be able to import the required components to test the application:

```
pip install -e .
pip install pytest
```

Run and watch the tests pass, within the top-level `flaskr/` directory as:

```
py.test
```

## Testing + setuptools

One way to handle testing is to integrate it with `setuptools`. Here that requires adding a couple of lines to the `setup.py` file and creating a new file `setup.cfg`. One benefit of running the tests this way is that you do not have to install `pytest`. Go ahead and update the `setup.py` file to contain:

```
from setuptools import setup

setup(
    name='flaskr',
    packages=['flaskr'],
    include_package_data=True,
    install_requires=[
        'flask',
    ],
    setup_requires=[
        'pytest-runner',
    ],
    tests_require=[
        'pytest',
    ],
)
```

Now create `setup.cfg` in the project root (alongside `setup.py`):

```
[aliases]
test=pytest
```

Now you can run:

```
python setup.py test
```

This calls on the alias created in `setup.cfg` which in turn runs `pytest` via `pytest-runner`, as the `setup.py` script has been called. (Recall the *setup_requires* argument in `setup.py`) Following the standard rules of test-discovery your tests will be found, run, and hopefully pass.

This is one possible way to run and manage testing. Here `pytest` is used, but there are other options such as `nose`. Integrating testing with `setuptools` is convenient because it is not necessary to actually download `pytest` or any other testing framework one might use.

CHAPTER 6

# Templates

Flask leverages Jinja2 as template engine. You are obviously free to use a different template engine, but you still have to install Jinja2 to run Flask itself. This requirement is necessary to enable rich extensions. An extension can depend on Jinja2 being present.

This section only gives a very quick introduction into how Jinja2 is integrated into Flask. If you want information on the template engine's syntax itself, head over to the official Jinja2 Template Documentation for more information.

## Jinja Setup

Unless customized, Jinja2 is configured by Flask as follows:

- autoescaping is enabled for all templates ending in `.html`, `.htm`, `.xml` as well as `.xhtml` when using `render_template()`.
- autoescaping is enabled for all strings when using `render_template_string()`.
- a template has the ability to opt in/out autoescaping with the `{% autoescape %}` tag.
- Flask inserts a couple of global functions and helpers into the Jinja2 context, additionally to the values that are present by default.

## Standard Context

The following global variables are available within Jinja2 templates by default:

**config**
: The current configuration object (`flask.config`)

  New in version 0.6.

  Changed in version 0.10: This is now always available, even in imported templates.

**request**
: The current request object (`flask.request`). This variable is unavailable if the template was rendered without an active request context.

**session**
: The current session object (`flask.session`). This variable is unavailable if the template was rendered without an active request context.

**g**
: The request-bound object for global variables (`flask.g`). This variable is unavailable if the template was rendered without an active request context.

**url_for()**
: The `flask.url_for()` function.

**get_flashed_messages()**
: The `flask.get_flashed_messages()` function.

---

**The Jinja Context Behavior**

These variables are added to the context of variables, they are not global variables. The difference is that by default these will not show up in the context of imported templates. This is partially caused by performance considerations, partially to keep things explicit.

What does this mean for you? If you have a macro you want to import, that needs to access the request object you have two possibilities:

1. you explicitly pass the request to the macro as parameter, or the attribute of the request object you are interested in.
2. you import the macro "with context".

Importing with context looks like this:

```
{% from '_helpers.html' import my_macro with context %}
```

---

# Standard Filters

These filters are available in Jinja2 additionally to the filters provided by Jinja2 itself:

**tojson**()
: This function converts the given object into JSON representation. This is for example very helpful if you try to generate JavaScript on the fly.

  Note that inside `script` tags no escaping must take place, so make sure to disable escaping with `|safe` before Flask 0.10 if you intend to use it inside `script` tags:

```
<script type=text/javascript>
    doSomethingWith({{ user.username|tojson|safe }});
</script>
```

## Controlling Autoescaping

Autoescaping is the concept of automatically escaping special characters for you. Special characters in the sense of HTML (or XML, and thus XHTML) are `&`, `>`, `<`, `"` as well as `'`. Because these characters carry specific meanings in documents on their own you have to replace them by so called "entities" if you want to use them for text. Not doing so would not only cause user frustration by the inability to use these characters in text, but can also lead to security problems. (see *Cross-Site Scripting (XSS)*)

Sometimes however you will need to disable autoescaping in templates. This can be the case if you want to explicitly inject HTML into pages, for example if they come from a system that generates secure HTML like a markdown to HTML converter.

There are three ways to accomplish that:

- In the Python code, wrap the HTML string in a `Markup` object before passing it to the template. This is in general the recommended way.
- Inside the template, use the `|safe` filter to explicitly mark a string as safe HTML (`{{ myvariable|safe }}`)
- Temporarily disable the autoescape system altogether.

To disable the autoescape system in templates, you can use the `{% autoescape %}` block:

```
{% autoescape false %}
    <p>autoescaping is disabled here
    <p>{{ will_not_be_escaped }}
{% endautoescape %}
```

Whenever you do this, please be very cautious about the variables you are using in this block.

## Registering Filters

If you want to register your own filters in Jinja2 you have two ways to do that. You can either put them by hand into the `jinja_env` of the application or use the

`template_filter()` decorator.

The two following examples work the same and both reverse an object:

```
@app.template_filter('reverse')
def reverse_filter(s):
    return s[::-1]

def reverse_filter(s):
    return s[::-1]
app.jinja_env.filters['reverse'] = reverse_filter
```

In case of the decorator the argument is optional if you want to use the function name as name of the filter. Once registered, you can use the filter in your templates in the same way as Jinja2's builtin filters, for example if you have a Python list in context called *mylist*:

```
{% for x in mylist | reverse %}
{% endfor %}
```

## Context Processors

To inject new variables automatically into the context of a template, context processors exist in Flask. Context processors run before the template is rendered and have the ability to inject new values into the template context. A context processor is a function that returns a dictionary. The keys and values of this dictionary are then merged with the template context, for all templates in the app:

```
@app.context_processor
def inject_user():
    return dict(user=g.user)
```

The context processor above makes a variable called *user* available in the template with the value of *g.user*. This example is not very interesting because *g* is available in templates anyways, but it gives an idea how this works.

Variables are not limited to values; a context processor can also make functions available to templates (since Python allows passing around functions):

```
@app.context_processor
def utility_processor():
    def format_price(amount, currency=u'€'):
        return u'{0:.2f}{1}'.format(amount, currency)
    return dict(format_price=format_price)
```

The context processor above makes the *format_price* function available to all templates:

```
{{ format_price(0.33) }}
```

You could also build *format_price* as a template filter (see *Registering Filters*), but this demonstrates how to pass functions in a context processor.

CHAPTER 7

# Testing Flask Applications

**Something that is untested is broken.**

The origin of this quote is unknown and while it is not entirely correct, it is also not far from the truth. Untested applications make it hard to improve existing code and developers of untested applications tend to become pretty paranoid. If an application has automated tests, you can safely make changes and instantly know if anything breaks.

Flask provides a way to test your application by exposing the Werkzeug test `Client` and handling the context locals for you. You can then use that with your favourite testing solution. In this documentation we will use the `unittest` package that comes pre-installed with Python.

## The Application

First, we need an application to test; we will use the application from the *Tutorial*. If you don't have that application yet, get the sources from the examples.

## The Testing Skeleton

In order to test the application, we add a second module (`flaskr_tests.py`) and create a unittest skeleton there:

```
import os
import flaskr
```

```
import unittest
import tempfile

class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        flaskr.app.testing = True
        self.app = flaskr.app.test_client()
        with flaskr.app.app_context():
            flaskr.init_db()

    def tearDown(self):
        os.close(self.db_fd)
        os.unlink(flaskr.app.config['DATABASE'])

if __name__ == '__main__':
    unittest.main()
```

The code in the `setUp()` method creates a new test client and initializes a new database. This function is called before each individual test function is run. To delete the database after the test, we close the file and remove it from the filesystem in the `tearDown()` method. Additionally during setup the `TESTING` config flag is activated. What it does is disable the error catching during request handling so that you get better error reports when performing test requests against the application.

This test client will give us a simple interface to the application. We can trigger test requests to the application, and the client will also keep track of cookies for us.

Because SQLite3 is filesystem-based we can easily use the tempfile module to create a temporary database and initialize it. The `mkstemp()` function does two things for us: it returns a low-level file handle and a random file name, the latter we use as database name. We just have to keep the *db_fd* around so that we can use the `os.close()` function to close the file.

If we now run the test suite, we should see the following output:

```
$ python flaskr_tests.py

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Even though it did not run any actual tests, we already know that our flaskr application is syntactically valid, otherwise the import would have died with an exception.

## The First Test

Now it's time to start testing the functionality of the application. Let's check that the application shows "No entries here so far" if we access the root of the application (/). To do this, we add a new test method to our class, like this:

```
class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        flaskr.app.testing = True
        self.app = flaskr.app.test_client()
        with flaskr.app.app_context():
            flaskr.init_db()

    def tearDown(self):
        os.close(self.db_fd)
        os.unlink(flaskr.app.config['DATABASE'])

    def test_empty_db(self):
        rv = self.app.get('/')
        assert b'No entries here so far' in rv.data
```

Notice that our test functions begin with the word *test*; this allows `unittest` to automatically identify the method as a test to run.

By using *self.app.get* we can send an HTTP `GET` request to the application with the given path. The return value will be a `response_class` object. We can now use the `data` attribute to inspect the return value (as string) from the application. In this case, we ensure that `'No entries here so far'` is part of the output.

Run it again and you should see one passing test:

```
$ python flaskr_tests.py
.
----------------------------------------------------------------------
Ran 1 test in 0.034s

OK
```

## Logging In and Out

The majority of the functionality of our application is only available for the administrative user, so we need a way to log our test client in and out of the application. To do this, we fire some requests to the login and logout pages with the required form data (username and password). And because the login and logout pages redirect, we tell the client to *follow_redirects*.

Add the following two methods to your *FlaskrTestCase* class:

```
def login(self, username, password):
    return self.app.post('/login', data=dict(
        username=username,
        password=password
    ), follow_redirects=True)

def logout(self):
    return self.app.get('/logout', follow_redirects=True)
```

Now we can easily test that logging in and out works and that it fails with invalid credentials. Add this new test to the class:

```
def test_login_logout(self):
    rv = self.login('admin', 'default')
    assert b'You were logged in' in rv.data
    rv = self.logout()
    assert b'You were logged out' in rv.data
    rv = self.login('adminx', 'default')
    assert b'Invalid username' in rv.data
    rv = self.login('admin', 'defaultx')
    assert b'Invalid password' in rv.data
```

## Test Adding Messages

We should also test that adding messages works. Add a new test method like this:

```
def test_messages(self):
    self.login('admin', 'default')
    rv = self.app.post('/add', data=dict(
        title='<Hello>',
        text='<strong>HTML</strong> allowed here'
    ), follow_redirects=True)
    assert b'No entries here so far' not in rv.data
    assert b'&lt;Hello&gt;' in rv.data
    assert b'<strong>HTML</strong> allowed here' in rv.data
```

Here we check that HTML is allowed in the text but not in the title, which is the intended behavior.

Running that should now give us three passing tests:

```
$ python flaskr_tests.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.332s

OK
```

For more complex tests with headers and status codes, check out the MiniTwit Example from the sources which contains a larger test suite.

## Other Testing Tricks

Besides using the test client as shown above, there is also the `test_request_context()` method that can be used in combination with the `with` statement to activate a request context temporarily. With this you can access the `request`, `g` and `session` objects like in view functions. Here is a full example that demonstrates this approach:

```
import flask

app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    assert flask.request.path == '/'
    assert flask.request.args['name'] == 'Peter'
```

All the other objects that are context bound can be used in the same way.

If you want to test your application with different configurations and there does not seem to be a good way to do that, consider switching to application factories (see *Application Factories*).

Note however that if you are using a test request context, the `before_request()` and `after_request()` functions are not called automatically. However `teardown_request()` functions are indeed executed when the test request context leaves the `with` block. If you do want the `before_request()` functions to be called as well, you need to call `preprocess_request()` yourself:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    app.preprocess_request()
    ...
```

This can be necessary to open database connections or something similar depending on how your application was designed.

If you want to call the `after_request()` functions you need to call into `process_response()` which however requires that you pass it a response object:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    resp = Response('...')
    resp = app.process_response(resp)
    ...
```

This in general is less useful because at that point you can directly start using the test client.

## Faking Resources and Context

New in version 0.10.

A very common pattern is to store user authorization information and database connections on the application context or the `flask.g` object. The general pattern for this is to put the object on there on first usage and then to remove it on a teardown. Imagine for instance this code to get the current user:

```
def get_user():
    user = getattr(g, 'user', None)
    if user is None:
        user = fetch_current_user_from_database()
        g.user = user
    return user
```

For a test it would be nice to override this user from the outside without having to change some code. This can be accomplished with hooking the `flask.appcontext_pushed` signal:

```
from contextlib import contextmanager
from flask import appcontext_pushed, g

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
    with appcontext_pushed.connected_to(handler, app):
        yield
```

And then to use it:

```
from flask import json, jsonify

@app.route('/users/me')
def users_me():
    return jsonify(username=g.user.username)

with user_set(app, my_user):
    with app.test_client() as c:
        resp = c.get('/users/me')
        data = json.loads(resp.data)
        self.assert_equal(data['username'], my_user.username)
```

## Keeping the Context Around

New in version 0.4.

Sometimes it is helpful to trigger a regular request but still keep the context around for a little longer so that additional introspection can happen. With Flask 0.4 this is possible by using the `test_client()` with a `with` block:

```
app = flask.Flask(__name__)

with app.test_client() as c:
    rv = c.get('/?tequila=42')
    assert request.args['tequila'] == '42'
```

If you were to use just the `test_client()` without the `with` block, the `assert` would fail with an error because *request* is no longer available (because you are trying to use it outside of the actual request).

## Accessing and Modifying Sessions

New in version 0.8.

Sometimes it can be very helpful to access or modify the sessions from the test client. Generally there are two ways for this. If you just want to ensure that a session has certain keys set to certain values you can just keep the context around and access `flask.session`:

```
with app.test_client() as c:
    rv = c.get('/')
    assert flask.session['foo'] == 42
```

This however does not make it possible to also modify the session or to access the session before a request was fired. Starting with Flask 0.8 we provide a so called "session transaction" which simulates the appropriate calls to open a session in the context of the test client and to modify it. At the end of the transaction the session is stored. This works independently of the session backend used:

```
with app.test_client() as c:
    with c.session_transaction() as sess:
        sess['a_key'] = 'a value'

    # once this is reached the session was stored
```

Note that in this case you have to use the `sess` object instead of the `flask.session` proxy. The object however itself will provide the same interface.

CHAPTER 8

# Application Errors

New in version 0.3.

Applications fail, servers fail. Sooner or later you will see an exception in production. Even if your code is 100% correct, you will still see exceptions from time to time. Why? Because everything else involved will fail. Here are some situations where perfectly fine code can lead to server errors:

- the client terminated the request early and the application was still reading from the incoming data
- the database server was overloaded and could not handle the query
- a filesystem is full
- a harddrive crashed
- a backend server overloaded
- a programming error in a library you are using
- network connection of the server to another system failed

And that's just a small sample of issues you could be facing. So how do we deal with that sort of problem? By default if your application runs in production mode, Flask will display a very simple page for you and log the exception to the `logger`.

But there is more you can do, and we will cover some better setups to deal with errors.

# Error Logging Tools

Sending error mails, even if just for critical ones, can become overwhelming if enough users are hitting the error and log files are typically never looked at. This is why we recommend using Sentry for dealing with application errors. It's available as an Open Source project on GitHub and is also available as a hosted version which you can try for free. Sentry aggregates duplicate errors, captures the full stack trace and local variables for debugging, and sends you mails based on new errors or frequency thresholds.

To use Sentry you need to install the *raven* client:

```
$ pip install raven
```

And then add this to your Flask app:

```
from raven.contrib.flask import Sentry
sentry = Sentry(app, dsn='YOUR_DSN_HERE')
```

Or if you are using factories you can also init it later:

```
from raven.contrib.flask import Sentry
sentry = Sentry(dsn='YOUR_DSN_HERE')

def create_app():
    app = Flask(__name__)
    sentry.init_app(app)
    ...
    return app
```

The *YOUR_DSN_HERE* value needs to be replaced with the DSN value you get from your Sentry installation.

Afterwards failures are automatically reported to Sentry and from there you can receive error notifications.

# Error handlers

You might want to show custom error pages to the user when an error occurs. This can be done by registering error handlers.

Error handlers are normal *Pluggable Views* but instead of being registered for routes, they are registered for exceptions that are raised while trying to do something else.

## Registering

Register error handlers using `errorhandler()` or `register_error_handler()`:

```
@app.errorhandler(werkzeug.exceptions.BadRequest)
def handle_bad_request(e):
    return 'bad request!'

app.register_error_handler(400, lambda e: 'bad request!')
```

Those two ways are equivalent, but the first one is more clear and leaves you with a function to call on your whim (and in tests). Note that `werkzeug.exceptions.HTTPException` subclasses like `BadRequest` from the example and their HTTP codes are interchangeable when handed to the registration methods or decorator (`BadRequest.code == 400`).

You are however not limited to `HTTPException` or HTTP status codes but can register a handler for every exception class you like.

Changed in version 0.11: Errorhandlers are now prioritized by specificity of the exception classes they are registered for instead of the order they are registered in.

### Handling

Once an exception instance is raised, its class hierarchy is traversed, and searched for in the exception classes for which handlers are registered. The most specific handler is selected.

E.g. if an instance of `ConnectionRefusedError` is raised, and a handler is registered for `ConnectionError` and `ConnectionRefusedError`, the more specific `ConnectionRefusedError` handler is called on the exception instance, and its response is shown to the user.

## Error Mails

If the application runs in production mode (which it will do on your server) you might not see any log messages. The reason for that is that Flask by default will just report to the WSGI error stream or stderr (depending on what's available). Where this ends up is sometimes hard to find. Often it's in your webserver's log files.

I can pretty much promise you however that if you only use a logfile for the application errors you will never look at it except for debugging an issue when a user reported it for you. What you probably want instead is a mail the second the exception happened. Then you get an alert and you can do something about it.

Flask uses the Python builtin logging system, and it can actually send you mails for errors which is probably what you want. Here is how you can configure the Flask logger to send you mails for exceptions:

```
ADMINS = ['yourname@example.com']
if not app.debug:
    import logging
```

```
    from logging.handlers import SMTPHandler
    mail_handler = SMTPHandler('127.0.0.1',
                               'server-error@example.com',
                               ADMINS, 'YourApplication Failed')
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
```

So what just happened? We created a new `SMTPHandler` that will send mails with the mail server listening on `127.0.0.1` to all the *ADMINS* from the address *server-error@example.com* with the subject "YourApplication Failed". If your mail server requires credentials, these can also be provided. For that check out the documentation for the `SMTPHandler`.

We also tell the handler to only send errors and more critical messages. Because we certainly don't want to get a mail for warnings or other useless logs that might happen during request handling.

Before you run that in production, please also look at *Controlling the Log Format* to put more information into that error mail. That will save you from a lot of frustration.

## Logging to a File

Even if you get mails, you probably also want to log warnings. It's a good idea to keep as much information around that might be required to debug a problem. By default as of Flask 0.11, errors are logged to your webserver's log automatically. Warnings however are not. Please note that Flask itself will not issue any warnings in the core system, so it's your responsibility to warn in the code if something seems odd.

There are a couple of handlers provided by the logging system out of the box but not all of them are useful for basic error logging. The most interesting are probably the following:

- `FileHandler` - logs messages to a file on the filesystem.
- `RotatingFileHandler` - logs messages to a file on the filesystem and will rotate after a certain number of messages.
- `NTEventLogHandler` - will log to the system event log of a Windows system. If you are deploying on a Windows box, this is what you want to use.
- `SysLogHandler` - sends logs to a UNIX syslog.

Once you picked your log handler, do like you did with the SMTP handler above, just make sure to use a lower setting (I would recommend *WARNING*):

```
if not app.debug:
    import logging
    from themodule import TheHandlerYouWant
    file_handler = TheHandlerYouWant(...)
    file_handler.setLevel(logging.WARNING)
    app.logger.addHandler(file_handler)
```

## Controlling the Log Format

By default a handler will only write the message string into a file or send you that message as mail. A log record stores more information, and it makes a lot of sense to configure your logger to also contain that information so that you have a better idea of why that error happened, and more importantly, where it did.

A formatter can be instantiated with a format string. Note that tracebacks are appended to the log entry automatically. You don't have to do that in the log formatter format string.

Here are some example setups:

### Email

```
from logging import Formatter
mail_handler.setFormatter(Formatter('''
Message type:       %(levelname)s
Location:           %(pathname)s:%(lineno)d
Module:             %(module)s
Function:           %(funcName)s
Time:               %(asctime)s

Message:

%(message)s
'''))
```

### File logging

```
from logging import Formatter
file_handler.setFormatter(Formatter(
    '%(asctime)s %(levelname)s: %(message)s '
    '[in %(pathname)s:%(lineno)d]'
))
```

### Complex Log Formatting

Here is a list of useful formatting variables for the format string. Note that this list is not complete, consult the official documentation of the `logging` package for a full list.

| Format | Description |
|---|---|
| `%(levelname)s` | Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`). |
| `%(pathname)s` | Full pathname of the source file where the logging call was issued (if available). |
| `%(filename)s` | Filename portion of pathname. |
| `%(module)s` | Module (name portion of filename). |
| `%(funcName)s` | Name of function containing the logging call. |
| `%(lineno)d` | Source line number where the logging call was issued (if available). |
| `%(asctime)s` | Human-readable time when the `LogRecord` was created. By default this is of the form `"2003-07-08 16:49:45,896"` (the numbers after the comma are millisecond portion of the time). This can be changed by subclassing the formatter and overriding the `formatTime()` method. |
| `%(message)s` | The logged message, computed as `msg % args` |

If you want to further customize the formatting, you can subclass the formatter. The formatter has three interesting methods:

**`format()`:** handles the actual formatting. It is passed a `LogRecord` object and has to return the formatted string.

**`formatTime()`:** called for *asctime* formatting. If you want a different time format you can override this method.

**`formatException()`** called for exception formatting. It is passed an `exc_info` tuple and has to return a string. The default is usually fine, you don't have to override it.

For more information, head over to the official documentation.

## Other Libraries

So far we only configured the logger your application created itself. Other libraries might log themselves as well. For example, SQLAlchemy uses logging heavily in its core. While there is a method to configure all loggers at once in the `logging` package, I would not recommend using it. There might be a situation in which you want to have multiple separate applications running side by side in the same Python interpreter and then it becomes impossible to have different logging setups for those.

Instead, I would recommend figuring out which loggers you are interested in, getting the loggers with the `getLogger()` function and iterating over them to attach handlers:

```
from logging import getLogger
loggers = [app.logger, getLogger('sqlalchemy'),
           getLogger('otherlibrary')]
for logger in loggers:
    logger.addHandler(mail_handler)
    logger.addHandler(file_handler)
```

CHAPTER 9

# Debugging Application Errors

For production applications, configure your application with logging and notifications as described in *Application Errors*. This section provides pointers when debugging deployment configuration and digging deeper with a full-featured Python debugger.

## When in Doubt, Run Manually

Having problems getting your application configured for production? If you have shell access to your host, verify that you can run your application manually from the shell in the deployment environment. Be sure to run under the same user account as the configured deployment to troubleshoot permission issues. You can use Flask's builtin development server with *debug=True* on your production host, which is helpful in catching configuration issues, but **be sure to do this temporarily in a controlled environment.** Do not run in production with *debug=True*.

## Working with Debuggers

To dig deeper, possibly to trace code execution, Flask provides a debugger out of the box (see *Debug Mode*). If you would like to use another Python debugger, note that debuggers interfere with each other. You have to set some options in order to use your favorite debugger:

- `debug` - whether to enable debug mode and catch exceptions
- `use_debugger` - whether to use the internal Flask debugger
- `use_reloader` - whether to reload and fork the process on exception

debug must be True (i.e., exceptions must be caught) in order for the other two options to have any value.

If you're using Aptana/Eclipse for debugging you'll need to set both `use_debugger` and `use_reloader` to False.

A possible useful pattern for configuration is to set the following in your config.yaml (change the block as appropriate for your application, of course):

```
FLASK:
    DEBUG: True
    DEBUG_WITH_APTANA: True
```

Then in your application's entry-point (main.py), you could have something like:

```
if __name__ == "__main__":
    # To allow aptana to receive errors, set use_debugger=False
    app = create_app(config="config.yaml")

    if app.debug: use_debugger = True
    try:
        # Disable Flask's debugger if external debugger is requested
        use_debugger = not(app.config.get('DEBUG_WITH_APTANA'))
    except:
        pass
    app.run(use_debugger=use_debugger, debug=app.debug,
            use_reloader=use_debugger, host='0.0.0.0')
```

CHAPTER 10

# Configuration Handling

New in version 0.3.

Applications need some kind of configuration. There are different settings you might want to change depending on the application environment like toggling the debug mode, setting the secret key, and other such environment-specific things.

The way Flask is designed usually requires the configuration to be available when the application starts up. You can hardcode the configuration in the code, which for many small applications is not actually that bad, but there are better ways.

Independent of how you load your config, there is a config object available which holds the loaded configuration values: The `config` attribute of the `Flask` object. This is the place where Flask itself puts certain configuration values and also where extensions can put their configuration values. But this is also where you can have your own configuration.

## Configuration Basics

The `config` is actually a subclass of a dictionary and can be modified just like any dictionary:

```
app = Flask(__name__)
app.config['DEBUG'] = True
```

Certain configuration values are also forwarded to the `Flask` object so you can read and write them from there:

```
app.debug = True
```

To update multiple keys at once you can use the `dict.update()` method:

```
app.config.update(
    DEBUG=True,
    SECRET_KEY='...'
)
```

## Builtin Configuration Values

The following configuration values are used internally by Flask:

| | |
|---|---|
| DEBUG | enable/disable debug mode |
| TESTING | enable/disable testing mode |
| PROPAGATE_EXCEPTIONS | explicitly enable or disable the propagation of exceptions. If not set or explicitly set to None this is implicitly true if either TESTING or DEBUG is true. |
| PRESERVE_CONTEXT_ON_EXCEPTION | By default if the application is in debug mode the request context is not popped on exceptions to enable debuggers to introspect the data. This can be disabled by this key. You can also use this setting to force-enable it for non debug execution which might be useful to debug production applications (but also very risky). |
| SECRET_KEY | the secret key |
| SESSION_COOKIE_NAME | the name of the session cookie |
| SESSION_COOKIE_DOMAIN | the domain for the session cookie. If this is not set, the cookie will be valid for all subdomains of SERVER_NAME. |
| SESSION_COOKIE_PATH | the path for the session cookie. If this is not set the cookie will be valid for all of APPLICATION_ROOT or if that is not set for '/'. |
| SESSION_COOKIE_HTTPONLY | controls if the cookie should be set with the httponly flag. Defaults to True. |
| SESSION_COOKIE_SECURE | controls if the cookie should be set with the secure flag. Defaults to False. |
| PERMANENT_SESSION_LIFETIME | the lifetime of a permanent session as datetime.timedelta object. Starting with Flask 0.8 this can also be an integer representing seconds. |
| SESSION_REFRESH_EACH_REQUEST | this flag controls how permanent sessions are refreshed. If set to True (which is the default) then the cookie is refreshed each request which automatically bumps the lifetime. If set to False a *set-cookie* header is only sent if the session is modified. Non permanent sessions are not affected by this. |
| USE_X_SENDFILE | enable/disable x-sendfile |
| LOGGER_NAME | the name of the logger |
| LOGGER_HANDLER_POLICY | the policy of the default logging handler. The default is 'always' which means that the default logging handler is always active. 'debug' will only activate logging in debug mode, 'production' will only log in production and 'never' disables it entirely. |
| SERVER_NAME | the name and port number of the server. Required for subdomain support (e.g.: 'myapp.dev:5000') Note that localhost does not support subdomains so setting this to "localhost" |
| | does not help. Setting a SERVER_NAME also by default enables URL generation without a request context but with an application context. |
| APPLICATION_ROOT | If the application does not occupy a whole do- |

**More on SERVER_NAME**

The SERVER_NAME key is used for the subdomain support. Because Flask cannot guess the subdomain part without the knowledge of the actual server name, this is required if you want to work with subdomains. This is also used for the session cookie.

Please keep in mind that not only Flask has the problem of not knowing what subdomains are, your web browser does as well. Most modern web browsers will not allow cross-subdomain cookies to be set on a server name without dots in it. So if your server name is `'localhost'` you will not be able to set a cookie for `'localhost'` and every subdomain of it. Please choose a different server name in that case, like `'myapplication.local'` and add this name + the subdomains you want to use into your host config or setup a local bind.

New in version 0.4: LOGGER_NAME

New in version 0.5: SERVER_NAME

New in version 0.6: MAX_CONTENT_LENGTH

New in version 0.7: PROPAGATE_EXCEPTIONS, PRESERVE_CONTEXT_ON_EXCEPTION

New in version 0.8: TRAP_BAD_REQUEST_ERRORS, TRAP_HTTP_EXCEPTIONS, APPLICATION_ROOT, SESSION_COOKIE_DOMAIN, SESSION_COOKIE_PATH, SESSION_COOKIE_HTTPONLY, SESSION_COOKIE_SECURE

New in version 0.9: PREFERRED_URL_SCHEME

New in version 0.10: JSON_AS_ASCII, JSON_SORT_KEYS, JSONIFY_PRETTYPRINT_REGULAR

New in version 0.11: SESSION_REFRESH_EACH_REQUEST, TEMPLATES_AUTO_RELOAD, LOGGER_HANDLER_POLICY, EXPLAIN_TEMPLATE_LOADING

## Configuring from Files

Configuration becomes more useful if you can store it in a separate file, ideally located outside the actual application package. This makes packaging and distributing your application possible via various package handling tools (*Deploying with Setuptools*) and finally modifying the configuration file afterwards.

So a common pattern is this:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

This first loads the configuration from the *yourapplication.default_settings* module and then overrides the values with the contents of the file the YOURAPPLICATION_SETTINGS environment variable points to. This environment variable can be set on Linux or OS X with the export command in the shell before starting the server:

```
$ export YOURAPPLICATION_SETTINGS=/path/to/settings.cfg
$ python run-app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader...
```

On Windows systems use the *set* builtin instead:

```
>set YOURAPPLICATION_SETTINGS=\path\to\settings.cfg
```

The configuration files themselves are actual Python files. Only values in uppercase are actually stored in the config object later on. So make sure to use uppercase letters for your config keys.

Here is an example of a configuration file:

```
# Example configuration
DEBUG = False
SECRET_KEY = '?\xbf,\xb4\x8d\xa3"<\x9c\xb0@\x0f5\xab,w\xee\x8d$0\x13\x8b83'
```

Make sure to load the configuration very early on, so that extensions have the ability to access the configuration when starting up. There are other methods on the config object as well to load from individual files. For a complete reference, read the `Config` object's documentation.

## Configuration Best Practices

The downside with the approach mentioned earlier is that it makes testing a little harder. There is no single 100% solution for this problem in general, but there are a couple of things you can keep in mind to improve that experience:

1. Create your application in a function and register blueprints on it. That way you can create multiple instances of your application with different configurations attached which makes unittesting a lot easier. You can use this to pass in configuration as needed.
2. Do not write code that needs the configuration at import time. If you limit yourself to request-only accesses to the configuration you can reconfigure the object later on as needed.

## Development / Production

Most applications need more than one configuration. There should be at least separate configurations for the production server and the one used during development. The easiest way to handle this is to use a default configuration that is always loaded and part of the version control, and a separate configuration that overrides the values as necessary as mentioned in the example above:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

Then you just have to add a separate `config.py` file and export `YOURAPPLICATION_SETTINGS=/path/to/config.py` and you are done. However there are alternative ways as well. For example you could use imports or subclassing.

What is very popular in the Django world is to make the import explicit in the config file by adding `from yourapplication.default_settings import *` to the top of the file and then overriding the changes by hand. You could also inspect an environment variable like `YOURAPPLICATION_MODE` and set that to *production*, *development* etc and import different hardcoded files based on that.

An interesting pattern is also to use classes and inheritance for configuration:

```
class Config(object):
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite://:memory:'

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
    DEBUG = True

class TestingConfig(Config):
    TESTING = True
```

To enable such a config you just have to call into `from_object()`:

```
app.config.from_object('configmodule.ProductionConfig')
```

There are many different ways and it's up to you how you want to manage your configuration files. However here a list of good recommendations:

- Keep a default configuration in version control. Either populate the config with this default configuration or import it in your own configuration files before overriding values.
- Use an environment variable to switch between the configurations. This can be done from outside the Python interpreter and makes development and deployment much easier because you can quickly and easily switch between different configs without having to touch the code at all. If you are working often on different projects you can even create your own script for sourcing that activates a virtualenv and exports the development configuration for you.
- Use a tool like fabric in production to push code and configurations separately to the production server(s). For some details about how to do that, head over to the *Deploying with Fabric* pattern.

# Instance Folders

New in version 0.8.

Flask 0.8 introduces instance folders. Flask for a long time made it possible to refer to paths relative to the application's folder directly (via `Flask.root_path`). This was also how many developers loaded configurations stored next to the application. Unfortunately however this only works well if applications are not packages in which case the root path refers to the contents of the package.

With Flask 0.8 a new attribute was introduced: `Flask.instance_path`. It refers to a new concept called the "instance folder". The instance folder is designed to not be under version control and be deployment specific. It's the perfect place to drop things that either change at runtime or configuration files.

You can either explicitly provide the path of the instance folder when creating the Flask application or you can let Flask autodetect the instance folder. For explicit configuration use the *instance_path* parameter:

```
app = Flask(__name__, instance_path='/path/to/instance/folder')
```

Please keep in mind that this path *must* be absolute when provided.

If the *instance_path* parameter is not provided the following default locations are used:

- Uninstalled module:

  ```
  /myapp.py
  /instance
  ```

- Uninstalled package:

  ```
  /myapp
      /__init__.py
  /instance
  ```

- Installed module or package:

  ```
  $PREFIX/lib/python2.X/site-packages/myapp
  $PREFIX/var/myapp-instance
  ```

  `$PREFIX` is the prefix of your Python installation. This can be `/usr` or the path to your virtualenv. You can print the value of `sys.prefix` to see what the prefix is set to.

Since the config object provided loading of configuration files from relative filenames we made it possible to change the loading via filenames to be relative to the instance path if wanted. The behavior of relative paths in config files can be flipped between "relative to the application root" (the default) to "relative to instance folder" via the *instance_relative_config* switch to the application constructor:

```
app = Flask(__name__, instance_relative_config=True)
```

Here is a full example of how to configure Flask to preload the config from a module and then override the config from a file in the config folder if it exists:

```
app = Flask(__name__, instance_relative_config=True)
app.config.from_object('yourapplication.default_settings')
app.config.from_pyfile('application.cfg', silent=True)
```

The path to the instance folder can be found via the `Flask.instance_path`. Flask also provides a shortcut to open a file from the instance folder with `Flask.open_instance_resource()`.

Example usage for both:

```
filename = os.path.join(app.instance_path, 'application.cfg')
with open(filename) as f:
    config = f.read()

# or via open_instance_resource:
with app.open_instance_resource('application.cfg') as f:
    config = f.read()
```

CHAPTER **11**

# Signals

New in version 0.6.

Starting with Flask 0.6, there is integrated support for signalling in Flask. This support is provided by the excellent blinker library and will gracefully fall back if it is not available.

What are signals? Signals help you decouple applications by sending notifications when actions occur elsewhere in the core framework or another Flask extensions. In short, signals allow certain senders to notify subscribers that something happened.

Flask comes with a couple of signals and other extensions might provide more. Also keep in mind that signals are intended to notify subscribers and should not encourage subscribers to modify data. You will notice that there are signals that appear to do the same thing like some of the builtin decorators do (eg: `request_started` is very similar to `before_request()`). However, there are differences in how they work. The core `before_request()` handler, for example, is executed in a specific order and is able to abort the request early by returning a response. In contrast all signal handlers are executed in undefined order and do not modify any data.

The big advantage of signals over handlers is that you can safely subscribe to them for just a split second. These temporary subscriptions are helpful for unittesting for example. Say you want to know what templates were rendered as part of a request: signals allow you to do exactly that.

## Subscribing to Signals

To subscribe to a signal, you can use the `connect()` method of a signal. The first argument is the function that should be called when the signal is emitted, the optional

second argument specifies a sender. To unsubscribe from a signal, you can use the `disconnect()` method.

For all core Flask signals, the sender is the application that issued the signal. When you subscribe to a signal, be sure to also provide a sender unless you really want to listen for signals from all applications. This is especially true if you are developing an extension.

For example, here is a helper context manager that can be used in a unittest to determine which templates were rendered and what variables were passed to the template:

```
from flask import template_rendered
from contextlib import contextmanager

@contextmanager
def captured_templates(app):
    recorded = []
    def record(sender, template, context, **extra):
        recorded.append((template, context))
    template_rendered.connect(record, app)
    try:
        yield recorded
    finally:
        template_rendered.disconnect(record, app)
```

This can now easily be paired with a test client:

```
with captured_templates(app) as templates:
    rv = app.test_client().get('/')
    assert rv.status_code == 200
    assert len(templates) == 1
    template, context = templates[0]
    assert template.name == 'index.html'
    assert len(context['items']) == 10
```

Make sure to subscribe with an extra `**extra` argument so that your calls don't fail if Flask introduces new arguments to the signals.

All the template rendering in the code issued by the application *app* in the body of the `with` block will now be recorded in the *templates* variable. Whenever a template is rendered, the template object as well as context are appended to it.

Additionally there is a convenient helper method (`connected_to()`) that allows you to temporarily subscribe a function to a signal with a context manager on its own. Because the return value of the context manager cannot be specified that way, you have to pass the list in as an argument:

```
from flask import template_rendered

def captured_templates(app, recorded, **extra):
    def record(sender, template, context):
        recorded.append((template, context))
```

```
    return template_rendered.connected_to(record, app)
```

The example above would then look like this:

```
templates = []
with captured_templates(app, templates, **extra):
    ...
    template, context = templates[0]
```

**Blinker API Changes**

The `connected_to()` method arrived in Blinker with version 1.1.

## Creating Signals

If you want to use signals in your own application, you can use the blinker library directly. The most common use case are named signals in a custom `Namespace`.. This is what is recommended most of the time:

```
from blinker import Namespace
my_signals = Namespace()
```

Now you can create new signals like this:

```
model_saved = my_signals.signal('model-saved')
```

The name for the signal here makes it unique and also simplifies debugging. You can access the name of the signal with the `name` attribute.

**For Extension Developers**

If you are writing a Flask extension and you want to gracefully degrade for missing blinker installations, you can do so by using the `flask.signals.Namespace` class.

## Sending Signals

If you want to emit a signal, you can do so by calling the `send()` method. It accepts a sender as first argument and optionally some keyword arguments that are forwarded to the signal subscribers:

```
class Model(object):
    ...
```

```
    def save(self):
        model_saved.send(self)
```

Try to always pick a good sender. If you have a class that is emitting a signal, pass `self` as sender. If you are emitting a signal from a random function, you can pass `current_app._get_current_object()` as sender.

---

**Passing Proxies as Senders**

Never pass `current_app` as sender to a signal. Use `current_app._get_current_object()` instead. The reason for this is that `current_app` is a proxy and not the real application object.

---

## Signals and Flask's Request Context

Signals fully support *The Request Context* when receiving signals. Context-local variables are consistently available between `request_started` and `request_finished`, so you can rely on `flask.g` and others as needed. Note the limitations described in *Sending Signals* and the `request_tearing_down` signal.

## Decorator Based Signal Subscriptions

With Blinker 1.1 you can also easily subscribe to signals by using the new `connect_via()` decorator:

```
from flask import template_rendered

@template_rendered.connect_via(app)
def when_template_rendered(sender, template, context, **extra):
    print 'Template %s is rendered with %s' % (template.name, context)
```

## Core Signals

Take a look at *Signals* for a list of all builtin signals.

CHAPTER 12

# Pluggable Views

New in version 0.7.

Flask 0.7 introduces pluggable views inspired by the generic views from Django which are based on classes instead of functions. The main intention is that you can replace parts of the implementations and this way have customizable pluggable views.

## Basic Principle

Consider you have a function that loads a list of objects from the database and renders into a template:

```
@app.route('/users/')
def show_users(page):
    users = User.query.all()
    return render_template('users.html', users=users)
```

This is simple and flexible, but if you want to provide this view in a generic fashion that can be adapted to other models and templates as well you might want more flexibility. This is where pluggable class-based views come into place. As the first step to convert this into a class based view you would do this:

```
from flask.views import View

class ShowUsers(View):

    def dispatch_request(self):
        users = User.query.all()
```

```
        return render_template('users.html', objects=users)

app.add_url_rule('/users/', view_func=ShowUsers.as_view('show_users'))
```

As you can see what you have to do is to create a subclass of `flask.views.View` and implement `dispatch_request()`. Then we have to convert that class into an actual view function by using the `as_view()` class method. The string you pass to that function is the name of the endpoint that view will then have. But this by itself is not helpful, so let's refactor the code a bit:

```
from flask.views import View

class ListView(View):

    def get_template_name(self):
        raise NotImplementedError()

    def render_template(self, context):
        return render_template(self.get_template_name(), **context)

    def dispatch_request(self):
        context = {'objects': self.get_objects()}
        return self.render_template(context)

class UserView(ListView):

    def get_template_name(self):
        return 'users.html'

    def get_objects(self):
        return User.query.all()
```

This of course is not that helpful for such a small example, but it's good enough to explain the basic principle. When you have a class-based view the question comes up what `self` points to. The way this works is that whenever the request is dispatched a new instance of the class is created and the `dispatch_request()` method is called with the parameters from the URL rule. The class itself is instantiated with the parameters passed to the `as_view()` function. For instance you can write a class like this:

```
class RenderTemplateView(View):
    def __init__(self, template_name):
        self.template_name = template_name
    def dispatch_request(self):
        return render_template(self.template_name)
```

And then you can register it like this:

```
app.add_url_rule('/about', view_func=RenderTemplateView.as_view(
    'about_page', template_name='about.html'))
```

## Method Hints

Pluggable views are attached to the application like a regular function by either using `route()` or better `add_url_rule()`. That however also means that you would have to provide the names of the HTTP methods the view supports when you attach this. In order to move that information to the class you can provide a `methods` attribute that has this information:

```
class MyView(View):
    methods = ['GET', 'POST']

    def dispatch_request(self):
        if request.method == 'POST':
            ...
        ...

app.add_url_rule('/myview', view_func=MyView.as_view('myview'))
```

## Method Based Dispatching

For RESTful APIs it's especially helpful to execute a different function for each HTTP method. With the `flask.views.MethodView` you can easily do that. Each HTTP method maps to a function with the same name (just in lowercase):

```
from flask.views import MethodView

class UserAPI(MethodView):

    def get(self):
        users = User.query.all()
        ...

    def post(self):
        user = User.from_form_data(request.form)
        ...

app.add_url_rule('/users/', view_func=UserAPI.as_view('users'))
```

That way you also don't have to provide the `methods` attribute. It's automatically set based on the methods defined in the class.

## Decorating Views

Since the view class itself is not the view function that is added to the routing system it does not make much sense to decorate the class itself. Instead you either have to decorate the return value of `as_view()` by hand:

```
def user_required(f):
    """Checks whether user is logged in or raises error 401."""
    def decorator(*args, **kwargs):
        if not g.user:
            abort(401)
        return f(*args, **kwargs)
    return decorator

view = user_required(UserAPI.as_view('users'))
app.add_url_rule('/users/', view_func=view)
```

Starting with Flask 0.8 there is also an alternative way where you can specify a list of decorators to apply in the class declaration:

```
class UserAPI(MethodView):
    decorators = [user_required]
```

Due to the implicit self from the caller's perspective you cannot use regular view decorators on the individual methods of the view however, keep this in mind.

## Method Views for APIs

Web APIs are often working very closely with HTTP verbs so it makes a lot of sense to implement such an API based on the `MethodView`. That said, you will notice that the API will require different URL rules that go to the same method view most of the time. For instance consider that you are exposing a user object on the web:

| URL | Method | Description |
|---|---|---|
| `/users/` | `GET` | Gives a list of all users |
| `/users/` | `POST` | Creates a new user |
| `/users/<id>` | `GET` | Shows a single user |
| `/users/<id>` | `PUT` | Updates a single user |
| `/users/<id>` | `DELETE` | Deletes a single user |

So how would you go about doing that with the `MethodView`? The trick is to take advantage of the fact that you can provide multiple rules to the same view.

Let's assume for the moment the view would look like this:

```
class UserAPI(MethodView):

    def get(self, user_id):
        if user_id is None:
            # return a list of users
            pass
        else:
            # expose a single user
            pass
```

```
    def post(self):
        # create a new user
        pass

    def delete(self, user_id):
        # delete a single user
        pass

    def put(self, user_id):
        # update a single user
        pass
```

So how do we hook this up with the routing system? By adding two rules and explicitly mentioning the methods for each:

```
user_view = UserAPI.as_view('user_api')
app.add_url_rule('/users/', defaults={'user_id': None},
                 view_func=user_view, methods=['GET',])
app.add_url_rule('/users/', view_func=user_view, methods=['POST',])
app.add_url_rule('/users/<int:user_id>', view_func=user_view,
                 methods=['GET', 'PUT', 'DELETE'])
```

If you have a lot of APIs that look similar you can refactor that registration code:

```
def register_api(view, endpoint, url, pk='id', pk_type='int'):
    view_func = view.as_view(endpoint)
    app.add_url_rule(url, defaults={pk: None},
                     view_func=view_func, methods=['GET',])
    app.add_url_rule(url, view_func=view_func, methods=['POST',])
    app.add_url_rule('%s<%s:%s>' % (url, pk_type, pk), view_func=view_func,
                     methods=['GET', 'PUT', 'DELETE'])

register_api(UserAPI, 'user_api', '/users/', pk='user_id')
```

CHAPTER 13

# The Application Context

New in version 0.9.

One of the design ideas behind Flask is that there are two different "states" in which code is executed. The application setup state in which the application implicitly is on the module level. It starts when the `Flask` object is instantiated, and it implicitly ends when the first request comes in. While the application is in this state a few assumptions are true:

- the programmer can modify the application object safely.
- no request handling happened so far
- you have to have a reference to the application object in order to modify it, there is no magic proxy that can give you a reference to the application object you're currently creating or modifying.

In contrast, during request handling, a couple of other rules exist:

- while a request is active, the context local objects (`flask.request` and others) point to the current request.
- any code can get hold of these objects at any time.

There is a third state which is sitting in between a little bit. Sometimes you are dealing with an application in a way that is similar to how you interact with applications during request handling; just that there is no request active. Consider, for instance, that you're sitting in an interactive Python shell and interacting with the application, or a command line application.

The application context is what powers the `current_app` context local.

## Purpose of the Application Context

The main reason for the application's context existence is that in the past a bunch of functionality was attached to the request context for lack of a better solution. Since one of the pillars of Flask's design is that you can have more than one application in the same Python process.

So how does the code find the "right" application? In the past we recommended passing applications around explicitly, but that caused issues with libraries that were not designed with that in mind.

A common workaround for that problem was to use the `current_app` proxy later on, which was bound to the current request's application reference. Since creating such a request context is an unnecessarily expensive operation in case there is no request around, the application context was introduced.

## Creating an Application Context

There are two ways to make an application context. The first one is implicit: whenever a request context is pushed, an application context will be created alongside if this is necessary. As a result, you can ignore the existence of the application context unless you need it.

The second way is the explicit way using the `app_context()` method:

```
from flask import Flask, current_app

app = Flask(__name__)
with app.app_context():
    # within this block, current_app points to app.
    print current_app.name
```

The application context is also used by the `url_for()` function in case a `SERVER_NAME` was configured. This allows you to generate URLs even in the absence of a request.

If no request context has been pushed and an application context has not been explicitly set, a `RuntimeError` will be raised.

```
RuntimeError: Working outside of application context.
```

## Locality of the Context

The application context is created and destroyed as necessary. It never moves between threads and it will not be shared between requests. As such it is the perfect place to store database connection information and other things. The internal stack object is called `flask._app_ctx_stack`. Extensions are free to store additional information on

the topmost level, assuming they pick a sufficiently unique name and should put their information there, instead of on the `flask.g` object which is reserved for user code.

For more information about that, see *Flask Extension Development*.

## Context Usage

The context is typically used to cache resources that need to be created on a per-request or usage case. For instance, database connections are destined to go there. When storing things on the application context unique names should be chosen as this is a place that is shared between Flask applications and extensions.

The most common usage is to split resource management into two parts:

1. an implicit resource caching on the context.
2. a context teardown based resource deallocation.

Generally there would be a `get_X()` function that creates resource `X` if it does not exist yet and otherwise returns the same resource, and a `teardown_X()` function that is registered as teardown handler.

This is an example that connects to a database:

```
import sqlite3
from flask import g

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = connect_to_database()
    return db

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

The first time `get_db()` is called the connection will be established. To make this implicit a `LocalProxy` can be used:

```
from werkzeug.local import LocalProxy
db = LocalProxy(get_db)
```

That way a user can directly access `db` which internally calls `get_db()`.

CHAPTER 14

# The Request Context

This document describes the behavior in Flask 0.7 which is mostly in line with the old behavior but has some small, subtle differences.

It is recommended that you read the *The Application Context* chapter first.

## Diving into Context Locals

Say you have a utility function that returns the URL the user should be redirected to. Imagine it would always redirect to the URL's `next` parameter or the HTTP referrer or the index page:

```
from flask import request, url_for

def redirect_url():
    return request.args.get('next') or \
           request.referrer or \
           url_for('index')
```

As you can see, it accesses the request object. If you try to run this from a plain Python shell, this is the exception you will see:

```
>>> redirect_url()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'request'
```

That makes a lot of sense because we currently do not have a request we could

access. So we have to make a request and bind it to the current context. The `test_request_context` method can create us a `RequestContext`:

```
>>> ctx = app.test_request_context('/?next=http://example.com/')
```

This context can be used in two ways. Either with the `with` statement or by calling the `push()` and `pop()` methods:

```
>>> ctx.push()
```

From that point onwards you can work with the request object:

```
>>> redirect_url()
u'http://example.com/'
```

Until you call *pop*:

```
>>> ctx.pop()
```

Because the request context is internally maintained as a stack you can push and pop multiple times. This is very handy to implement things like internal redirects.

For more information of how to utilize the request context from the interactive Python shell, head over to the *Working with the Shell* chapter.

## How the Context Works

If you look into how the Flask WSGI application internally works, you will find a piece of code that looks very much like this:

```
def wsgi_app(self, environ):
    with self.request_context(environ):
        try:
            response = self.full_dispatch_request()
        except Exception as e:
            response = self.make_response(self.handle_exception(e))
        return response(environ, start_response)
```

The method `request_context()` returns a new `RequestContext` object and uses it in combination with the `with` statement to bind the context. Everything that is called from the same thread from this point onwards until the end of the `with` statement will have access to the request globals (`flask.request` and others).

The request context internally works like a stack: The topmost level on the stack is the current active request. `push()` adds the context to the stack on the very top, `pop()` removes it from the stack again. On popping the application's `teardown_request()` functions are also executed.

Another thing of note is that the request context will automatically also create an *application context* when it's pushed and there is no application context for that application

so far.

## Callbacks and Errors

What happens if an error occurs in Flask during request processing? This particular behavior changed in 0.7 because we wanted to make it easier to understand what is actually happening. The new behavior is quite simple:

1. Before each request, `before_request()` functions are executed. If one of these functions return a response, the other functions are no longer called. In any case however the return value is treated as a replacement for the view's return value.
2. If the `before_request()` functions did not return a response, the regular request handling kicks in and the view function that was matched has the chance to return a response.
3. The return value of the view is then converted into an actual response object and handed over to the `after_request()` functions which have the chance to replace it or modify it in place.
4. At the end of the request the `teardown_request()` functions are executed. This always happens, even in case of an unhandled exception down the road or if a before-request handler was not executed yet or at all (for example in test environments sometimes you might want to not execute before-request callbacks).

Now what happens on errors? In production mode if an exception is not caught, the 500 internal server handler is called. In development mode however the exception is not further processed and bubbles up to the WSGI server. That way things like the interactive debugger can provide helpful debug information.

An important change in 0.7 is that the internal server error is now no longer post processed by the after request callbacks and after request callbacks are no longer guaranteed to be executed. This way the internal dispatching code looks cleaner and is easier to customize and understand.

The new teardown functions are supposed to be used as a replacement for things that absolutely need to happen at the end of request.

## Teardown Callbacks

The teardown callbacks are special callbacks in that they are executed at a different point. Strictly speaking they are independent of the actual request handling as they are bound to the lifecycle of the `RequestContext` object. When the request context is popped, the `teardown_request()` functions are called.

This is important to know if the life of the request context is prolonged by using the test client in a with statement or when using the request context from the command line:

```
with app.test_client() as client:
    resp = client.get('/foo')
    # the teardown functions are still not called at that point
    # even though the response ended and you have the response
    # object in your hand

# only when the code reaches this point the teardown functions
# are called.  Alternatively the same thing happens if another
# request was triggered from the test client
```

It's easy to see the behavior from the command line:

```
>>> app = Flask(__name__)
>>> @app.teardown_request
... def teardown_request(exception=None):
...     print 'this runs after request'
...
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> ctx.pop()
this runs after request
>>>
```

Keep in mind that teardown callbacks are always executed, even if before-request callbacks were not executed yet but an exception happened. Certain parts of the test system might also temporarily create a request context without calling the before-request handlers. Make sure to write your teardown-request handlers in a way that they will never fail.

## Notes On Proxies

Some of the objects provided by Flask are proxies to other objects. The reason behind this is that these proxies are shared between threads and they have to dispatch to the actual object bound to a thread behind the scenes as necessary.

Most of the time you don't have to care about that, but there are some exceptions where it is good to know that this object is an actual proxy:

- The proxy objects do not fake their inherited types, so if you want to perform actual instance checks, you have to do that on the instance that is being proxied (see *_get_current_object* below).
- if the object reference is important (so for example for sending *Signals*)

If you need to get access to the underlying object that is proxied, you can use the `_get_current_object()` method:

```
app = current_app._get_current_object()
my_signal.send(app)
```

## Context Preservation on Error

If an error occurs or not, at the end of the request the request context is popped and all data associated with it is destroyed. During development however that can be problematic as you might want to have the information around for a longer time in case an exception occurred. In Flask 0.6 and earlier in debug mode, if an exception occurred, the request context was not popped so that the interactive debugger can still provide you with important information.

Starting with Flask 0.7 you have finer control over that behavior by setting the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable. By default it's linked to the setting of `DEBUG`. If the application is in debug mode the context is preserved, in production mode it's not.

Do not force activate `PRESERVE_CONTEXT_ON_EXCEPTION` in production mode as it will cause your application to leak memory on exceptions. However it can be useful during development to get the same error preserving behavior as in development mode when attempting to debug an error that only occurs under production settings.

CHAPTER **15**

# Modular Applications with Blueprints

New in version 0.7.

Flask uses a concept of *blueprints* for making application components and supporting common patterns within an application or across applications. Blueprints can greatly simplify how large applications work and provide a central means for Flask extensions to register operations on applications. A `Blueprint` object works similarly to a `Flask` application object, but it is not actually an application. Rather it is a *blueprint* of how to construct or extend an application.

## Why Blueprints?

Blueprints in Flask are intended for these cases:

- Factor an application into a set of blueprints. This is ideal for larger applications; a project could instantiate an application object, initialize several extensions, and register a collection of blueprints.
- Register a blueprint on an application at a URL prefix and/or subdomain. Parameters in the URL prefix/subdomain become common view arguments (with defaults) across all view functions in the blueprint.
- Register a blueprint multiple times on an application with different URL rules.
- Provide template filters, static files, templates, and other utilities through blueprints. A blueprint does not have to implement applications or view functions.
- Register a blueprint on an application for any of these cases when initializing a Flask extension.

A blueprint in Flask is not a pluggable app because it is not actually an application – it's a set of operations which can be registered on an application, even multiple times. Why not have multiple application objects? You can do that (see *Application Dispatching*), but your applications will have separate configs and will be managed at the WSGI layer.

Blueprints instead provide separation at the Flask level, share application config, and can change an application object as necessary with being registered. The downside is that you cannot unregister a blueprint once an application was created without having to destroy the whole application object.

## The Concept of Blueprints

The basic concept of blueprints is that they record operations to execute when registered on an application. Flask associates view functions with blueprints when dispatching requests and generating URLs from one endpoint to another.

## My First Blueprint

This is what a very basic blueprint looks like. In this case we want to implement a blueprint that does simple rendering of static templates:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint('simple_page', __name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template('pages/%s.html' % page)
    except TemplateNotFound:
        abort(404)
```

When you bind a function with the help of the `@simple_page.route` decorator the blueprint will record the intention of registering the function *show* on the application when it's later registered. Additionally it will prefix the endpoint of the function with the name of the blueprint which was given to the `Blueprint` constructor (in this case also `simple_page`).

## Registering Blueprints

So how do you register that blueprint? Like this:

```
from flask import Flask
from yourapplication.simple_page import simple_page

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

If you check the rules registered on the application, you will find these:

```
[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

The first one is obviously from the application itself for the static files. The other two are for the *show* function of the `simple_page` blueprint. As you can see, they are also prefixed with the name of the blueprint and separated by a dot (`.`).

Blueprints however can also be mounted at different locations:

```
app.register_blueprint(simple_page, url_prefix='/pages')
```

And sure enough, these are the generated rules:

```
[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

On top of that you can register blueprints multiple times though not every blueprint might respond properly to that. In fact it depends on how the blueprint is implemented if it can be mounted more than once.

# Blueprint Resources

Blueprints can provide resources as well. Sometimes you might want to introduce a blueprint only for the resources it provides.

## Blueprint Resource Folder

Like for regular applications, blueprints are considered to be contained in a folder. While multiple blueprints can originate from the same folder, it does not have to be the case and it's usually not recommended.

The folder is inferred from the second argument to `Blueprint` which is usually *__name__*. This argument specifies what logical Python module or package corresponds to the blueprint. If it points to an actual Python package that package (which is a folder on the filesystem) is the resource folder. If it's a module, the package the module is contained in will be the resource folder. You can access the `Blueprint.root_path` property to see what the resource folder is:

```
>>> simple_page.root_path
'/Users/username/TestProject/yourapplication'
```

To quickly open sources from this folder you can use the `open_resource()` function:

```
with simple_page.open_resource('static/style.css') as f:
    code = f.read()
```

## Static Files

A blueprint can expose a folder with static files by providing a path to a folder on the filesystem via the *static_folder* keyword argument. It can either be an absolute path or one relative to the folder of the blueprint:

```
admin = Blueprint('admin', __name__, static_folder='static')
```

By default the rightmost part of the path is where it is exposed on the web. Because the folder is called `static` here it will be available at the location of the blueprint + `/static`. Say the blueprint is registered for `/admin` the static folder will be at `/admin/static`.

The endpoint is named *blueprint_name.static* so you can generate URLs to it like you would do to the static folder of the application:

```
url_for('admin.static', filename='style.css')
```

## Templates

If you want the blueprint to expose templates you can do that by providing the *template_folder* parameter to the `Blueprint` constructor:

```
admin = Blueprint('admin', __name__, template_folder='templates')
```

For static files, the path can be absolute or relative to the blueprint resource folder.

The template folder is added to the search path of templates but with a lower priority than the actual application's template folder. That way you can easily override templates that a blueprint provides in the actual application. This also means that if you don't want a blueprint template to be accidentally overridden, make sure that no other blueprint or actual application template has the same relative path. When multiple blueprints provide the same relative template path the first blueprint registered takes precedence over the others.

So if you have a blueprint in the folder `yourapplication/admin` and you want to render the template `'admin/index.html'` and you have provided `templates` as a *template_folder* you will have to create a file like this: `yourapplication/admin/templates/admin/index.html`. The reason for the extra `admin` folder is to avoid getting our tem-

plate overridden by a template named `index.html` in the actual application template folder.

To further reiterate this: if you have a blueprint named `admin` and you want to render a template called `index.html` which is specific to this blueprint, the best idea is to lay out your templates like this:

```
yourpackage/
    blueprints/
        admin/
            templates/
                admin/
                    index.html
            __init__.py
```

And then when you want to render the template, use `admin/index.html` as the name to look up the template by. If you encounter problems loading the correct templates enable the `EXPLAIN_TEMPLATE_LOADING` config variable which will instruct Flask to print out the steps it goes through to locate templates on every `render_template` call.

## Building URLs

If you want to link from one page to another you can use the `url_for()` function just like you normally would do just that you prefix the URL endpoint with the name of the blueprint and a dot (`.`):

```
url_for('admin.index')
```

Additionally if you are in a view function of a blueprint or a rendered template and you want to link to another endpoint of the same blueprint, you can use relative redirects by prefixing the endpoint with a dot only:

```
url_for('.index')
```

This will link to `admin.index` for instance in case the current request was dispatched to any other admin blueprint endpoint.

## Error Handlers

Blueprints support the errorhandler decorator just like the `Flask` application object, so it is easy to make Blueprint-specific custom error pages.

Here is an example for a "404 Page Not Found" exception:

```
@simple_page.errorhandler(404)
def page_not_found(e):
    return render_template('pages/404.html')
```

More information on error handling see *Custom Error Pages*.

CHAPTER **16**

# Flask Extensions

Flask extensions extend the functionality of Flask in various different ways. For instance they add support for databases and other common tasks.

## Finding Extensions

Flask extensions are listed on the Flask Extension Registry and can be downloaded with **`easy_install`** or **`pip`**. If you add a Flask extension as dependency to your `requirements.txt` or `setup.py` file they are usually installed with a simple command or when your application installs.

## Using Extensions

Extensions typically have documentation that goes along that shows how to use it. There are no general rules in how extensions are supposed to behave but they are imported from common locations. If you have an extension called `Flask-Foo` or `Foo-Flask` it should be always importable from `flask_foo`:

```
import flask_foo
```

## Building Extensions

While Flask Extension Registry contains many Flask extensions, you may not find an extension that fits your need. If this is the case, you can always create your own. Consider reading *Flask Extension Development* to develop your own Flask extension.

## Flask Before 0.8

If you are using Flask 0.7 or earlier the `flask.ext` package will not exist, instead you have to import from `flaskext.foo` or `flask_foo` depending on how the extension is distributed. If you want to develop an application that supports Flask 0.7 or earlier you should still import from the `flask.ext` package. We provide you with a compatibility module that provides this package for older versions of Flask. You can download it from GitHub: flaskext_compat.py

And here is how you can use it:

```
import flaskext_compat
flaskext_compat.activate()

from flask.ext import foo
```

Once the `flaskext_compat` module is activated the `flask.ext` will exist and you can start importing from there.

CHAPTER **17**

# Command Line Interface

New in version 0.11.

One of the nice new features in Flask 0.11 is the built-in integration of the click command line interface. This enables a wide range of new features for the Flask ecosystem and your own applications.

## Basic Usage

After installation of Flask you will now find a **`flask`** script installed into your virtualenv. If you don't want to install Flask or you have a special use-case you can also use `python -m flask` to accomplish exactly the same.

The way this script works is by providing access to all the commands on your Flask application's `Flask.cli` instance as well as some built-in commands that are always there. Flask extensions can also register more commands there if they desire so.

For the **`flask`** script to work, an application needs to be discovered. This is achieved by exporting the `FLASK_APP` environment variable. It can be either set to an import path or to a filename of a Python module that contains a Flask application.

In that imported file the name of the app needs to be called `app` or optionally be specified after a colon. For instance `mymodule:application` would tell it to use the *application* object in the `mymodule.py` file.

Given a `hello.py` file with the application in it named `app` this is how it can be run.

Environment variables (On Windows use `set` instead of `export`):

```
export FLASK_APP=hello
flask run
```

Or with a filename:

```
export FLASK_APP=/path/to/hello.py
flask run
```

## Virtualenv Integration

If you are constantly working with a virtualenv you can also put the `export FLASK_APP` into your `activate` script by adding it to the bottom of the file. That way every time you activate your virtualenv you automatically also activate the correct application name.

## Debug Flag

The **`flask`** script can also be instructed to enable the debug mode of the application automatically by exporting `FLASK_DEBUG`. If set to `1` debug is enabled or `0` disables it:

```
export FLASK_DEBUG=1
```

## Running a Shell

To run an interactive Python shell you can use the `shell` command:

```
flask shell
```

This will start up an interactive Python shell, setup the correct application context and setup the local variables in the shell. This is done by invoking the `Flask.make_shell_context()` method of the application. By default you have access to your `app` and `g`.

## Custom Commands

If you want to add more commands to the shell script you can do this easily. Flask uses click for the command interface which makes creating custom commands very easy. For instance if you want a shell command to initialize the database you can do this:

```
import click
from flask import Flask

app = Flask(__name__)

@app.cli.command()
def initdb():
    """Initialize the database."""
    click.echo('Init the db')
```

The command will then show up on the command line:

```
$ flask initdb
Init the db
```

## Application Context

Most commands operate on the application so it makes a lot of sense if they have the application context setup. Because of this, if you register a callback on `app.cli` with the `command()` the callback will automatically be wrapped through `cli.with_appcontext()` which informs the cli system to ensure that an application context is set up. This behavior is not available if a command is added later with `add_command()` or through other means.

It can also be disabled by passing `with_appcontext=False` to the decorator:

```
@app.cli.command(with_appcontext=False)
def example():
    pass
```

## Factory Functions

In case you are using factory functions to create your application (see *Application Factories*) you will discover that the **`flask`** command cannot work with them directly. Flask won't be able to figure out how to instantiate your application properly by itself. Because of this reason the recommendation is to create a separate file that instantiates applications. This is not the only way to make this work. Another is the *Custom Scripts* support.

For instance if you have a factory function that creates an application from a filename you could make a separate file that creates such an application from an environment variable.

This could be a file named `autoapp.py` with these contents:

```
import os
from yourapplication import create_app
app = create_app(os.environ['YOURAPPLICATION_CONFIG'])
```

Once this has happened you can make the **flask** command automatically pick it up:

```
export YOURAPPLICATION_CONFIG=/path/to/config.cfg
export FLASK_APP=/path/to/autoapp.py
```

From this point onwards **flask** will find your application.

## Custom Scripts

While the most common way is to use the **flask** command, you can also make your own "driver scripts". Since Flask uses click for the scripts there is no reason you cannot hook these scripts into any click application. There is one big caveat and that is, that commands registered to `Flask.cli` will expect to be (indirectly at least) launched from a `flask.cli.FlaskGroup` click group. This is necessary so that the commands know which Flask application they have to work with.

To understand why you might want custom scripts you need to understand how click finds and executes the Flask application. If you use the **flask** script you specify the application to work with on the command line or environment variable as an import name. This is simple but it has some limitations. Primarily it does not work with application factory functions (see *Application Factories*).

With a custom script you don't have this problem as you can fully customize how the application will be created. This is very useful if you write reusable applications that you want to ship to users and they should be presented with a custom management script.

To explain all of this, here is an example `manage.py` script that manages a hypothetical wiki application. We will go through the details afterwards:

```
import os
import click
from flask.cli import FlaskGroup

def create_wiki_app(info):
    from yourwiki import create_app
    return create_app(
        config=os.environ.get('WIKI_CONFIG', 'wikiconfig.py'))

@click.group(cls=FlaskGroup, create_app=create_wiki_app)
def cli():
    """This is a management script for the wiki application."""

if __name__ == '__main__':
    cli()
```

That's a lot of code for not much, so let's go through all parts step by step.

1. First we import the `click` library as well as the click extensions from the `flask.cli` package. Primarily we are here interested in the `FlaskGroup` click group.
2. The next thing we do is defining a function that is invoked with the script info object (`ScriptInfo`) from Flask and its purpose is to fully import and create the application. This can either directly import an application object or create it (see *Application Factories*). In this case we load the config from an environment variable.
3. Next step is to create a `FlaskGroup`. In this case we just make an empty function with a help doc string that just does nothing and then pass the `create_wiki_app` function as a factory function.

   Whenever click now needs to operate on a Flask application it will call that function with the script info and ask for it to be created.
4. All is rounded up by invoking the script.

## CLI Plugins

Flask extensions can always patch the `Flask.cli` instance with more commands if they want. However there is a second way to add CLI plugins to Flask which is through `setuptools`. If you make a Python package that should export a Flask command line plugin you can ship a `setup.py` file that declares an entrypoint that points to a click command:

Example `setup.py`:

```
from setuptools import setup

setup(
    name='flask-my-extension',
    ...
    entry_points='''
        [flask.commands]
        my-command=mypackage.commands:cli
    ''',
)
```

Inside `mypackage/commands.py` you can then export a Click object:

```
import click

@click.command()
def cli():
    """This is an example command."""
```

Once that package is installed in the same virtualenv as Flask itself you can run `flask my-command` to invoke your command. This is useful to provide extra functionality that

Flask itself cannot ship.

CHAPTER **18**

# Development Server

Starting with Flask 0.11 there are multiple built-in ways to run a development server. The best one is the **`flask`** command line utility but you can also continue using the `Flask.run()` method.

## Command Line

The **`flask`** command line script (*Command Line Interface*) is strongly recommended for development because it provides a superior reload experience due to how it loads the application. The basic usage is like this:

```
$ export FLASK_APP=my_application
$ export FLASK_DEBUG=1
$ flask run
```

This will enable the debugger, the reloader and then start the server on *http://localhost:5000/*.

The individual features of the server can be controlled by passing more arguments to the `run` option. For instance the reloader can be disabled:

```
$ flask run --no-reload
```

## In Code

The alternative way to start the application is through the `Flask.run()` method. This will immediately launch a local server exactly the same way the **`flask`** script does.

Example:

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

This works well for the common case but it does not work well for development which is why from Flask 0.11 onwards the **`flask`** method is recommended. The reason for this is that due to how the reload mechanism works there are some bizarre side-effects (like executing certain code twice, sometimes crashing without message or dying when a syntax or import error happens).

It is however still a perfectly valid method for invoking a non automatic reloading application.

CHAPTER **19**

# Working with the Shell

New in version 0.3.

One of the reasons everybody loves Python is the interactive shell. It basically allows you to execute Python commands in real time and immediately get results back. Flask itself does not come with an interactive shell, because it does not require any specific setup upfront, just import your application and start playing around.

There are however some handy helpers to make playing around in the shell a more pleasant experience. The main issue with interactive console sessions is that you're not triggering a request like a browser does which means that `g`, `request` and others are not available. But the code you want to test might depend on them, so what can you do?

This is where some helper functions come in handy. Keep in mind however that these functions are not only there for interactive shell usage, but also for unittesting and other situations that require a faked request context.

Generally it's recommended that you read the *The Request Context* chapter of the documentation first.

## Command Line Interface

Starting with Flask 0.11 the recommended way to work with the shell is the `flask shell` command which does a lot of this automatically for you. For instance the shell is automatically initialized with a loaded application context.

For more information see *Command Line Interface*.

## Creating a Request Context

The easiest way to create a proper request context from the shell is by using the `test_request_context` method which creates us a `RequestContext`:

```
>>> ctx = app.test_request_context()
```

Normally you would use the `with` statement to make this request object active, but in the shell it's easier to use the `push()` and `pop()` methods by hand:

```
>>> ctx.push()
```

From that point onwards you can work with the request object until you call *pop*:

```
>>> ctx.pop()
```

## Firing Before/After Request

By just creating a request context, you still don't have run the code that is normally run before a request. This might result in your database being unavailable if you are connecting to the database in a before-request callback or the current user not being stored on the g object etc.

This however can easily be done yourself. Just call `preprocess_request()`:

```
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> app.preprocess_request()
```

Keep in mind that the `preprocess_request()` function might return a response object, in that case just ignore it.

To shutdown a request, you need to trick a bit before the after request functions (triggered by `process_response()`) operate on a response object:

```
>>> app.process_response(app.response_class())
<Response 0 bytes [200 OK]>
>>> ctx.pop()
```

The functions registered as `teardown_request()` are automatically called when the context is popped. So this is the perfect place to automatically tear down resources that were needed by the request context (such as database connections).

## Further Improving the Shell Experience

If you like the idea of experimenting in a shell, create yourself a module with stuff you want to star import into your interactive session. There you could also define some

more helper methods for common things such as initializing the database, dropping tables etc.

Just put them into a module (like *shelltools*) and import from there:

```
>>> from shelltools import *
```

CHAPTER **20**

# Patterns for Flask

Certain things are common enough that the chances are high you will find them in most web applications. For example quite a lot of applications are using relational databases and user authentication. In that case, chances are they will open a database connection at the beginning of the request and get the information of the currently logged in user. At the end of the request, the database connection is closed again.

There are more user contributed snippets and patterns in the Flask Snippet Archives.

## Larger Applications

For larger applications it's a good idea to use a package instead of a module. That is quite simple. Imagine a small application looks like this:

```
/yourapplication
    yourapplication.py
    /static
        style.css
    /templates
        layout.html
        index.html
        login.html
        ...
```

## Simple Packages

To convert that into a larger one, just create a new folder `yourapplication` inside the existing one and move everything below it. Then rename `yourapplication.py` to `__init__.py`. (Make sure to delete all `.pyc` files first, otherwise things would most likely break)

You should then end up with something like that:

```
/yourapplication
    /yourapplication
        __init__.py
        /static
            style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

But how do you run your application now? The naive `python yourapplication/__init__.py` will not work. Let's just say that Python does not want modules in packages to be the startup file. But that is not a big problem, just add a new file called `setup.py` next to the inner `yourapplication` folder with the following contents:

```
from setuptools import setup

setup(
    name='yourapplication',
    packages=['yourapplication'],
    include_package_data=True,
    install_requires=[
        'flask',
    ],
)
```

In order to run the application you need to export an environment variable that tells Flask where to find the application instance:

```
export FLASK_APP=yourapplication
```

If you are outside of the project directory make sure to provide the exact path to your application directory. Similiarly you can turn on "debug mode" with this environment variable:

```
export FLASK_DEBUG=true
```

In order to install and run the application you need to issue the following commands:

```
pip install -e .
flask run
```

What did we gain from this? Now we can restructure the application a bit into multiple modules. The only thing you have to remember is the following quick checklist:

1. the *Flask* application object creation has to be in the `__init__.py` file. That way each module can import it safely and the *__name__* variable will resolve to the correct package.
2. all the view functions (the ones with a `route()` decorator on top) have to be imported in the `__init__.py` file. Not the object itself, but the module it is in. Import the view module **after the application object is created**.

Here's an example `__init__.py`:

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

import yourapplication.views
```

And this is what `views.py` would look like:

```
from yourapplication import app

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

You should then end up with something like that:

```
/yourapplication
    setup.py
    /yourapplication
        __init__.py
        views.py
        /static
            style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

### Circular Imports

Every Python programmer hates them, and yet we just added some: circular imports (That's when two modules depend on each other. In this case `views.py` depends on `__init__.py`). Be advised that this is a bad idea in general but here it is actually fine. The reason for this is that we are not actually using the views in `__init__.py` and just ensuring the module is imported and we are doing that at the bottom of the file.

There are still some problems with that approach but if you want to use decorators there is no way around that. Check out the *Becoming Big* section for some inspiration

how to deal with that.

### Working with Blueprints

If you have larger applications it's recommended to divide them into smaller groups where each group is implemented with the help of a blueprint. For a gentle introduction into this topic refer to the *Modular Applications with Blueprints* chapter of the documentation.

## Application Factories

If you are already using packages and blueprints for your application (*Modular Applications with Blueprints*) there are a couple of really nice ways to further improve the experience. A common pattern is creating the application object when the blueprint is imported. But if you move the creation of this object into a function, you can then create multiple instances of this app later.

So why would you want to do this?

1. Testing. You can have instances of the application with different settings to test every case.
2. Multiple instances. Imagine you want to run different versions of the same application. Of course you could have multiple instances with different configs set up in your webserver, but if you use factories, you can have multiple instances of the same application running in the same application process which can be handy.

So how would you then actually implement that?

### Basic Factories

The idea is to set up the application in a function. Like this:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.model import db
    db.init_app(app)

    from yourapplication.views.admin import admin
    from yourapplication.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)
```

```
    return app
```

The downside is that you cannot use the application object in the blueprints at import time. You can however use it from within a request. How do you get access to the application with the config? Use `current_app`:

```
from flask import current_app, Blueprint, render_template
admin = Blueprint('admin', __name__, url_prefix='/admin')

@admin.route('/')
def index():
    return render_template(current_app.config['INDEX_TEMPLATE'])
```

Here we look up the name of a template in the config.

## Factories & Extensions

It's preferable to create your extensions and app factories so that the extension object does not initially get bound to the application.

Using Flask-SQLAlchemy, as an example, you should not do something along those lines:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    db = SQLAlchemy(app)
```

But, rather, in model.py (or equivalent):

```
db = SQLAlchemy()
```

and in your application.py (or equivalent):

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.model import db
    db.init_app(app)
```

Using this design pattern, no application-specific state is stored on the extension object, so one extension object can be used for multiple apps. For more information about the design of extensions refer to *Flask Extension Development*.

### Using Applications

So to use such an application you then have to create the application first in a separate file otherwise the **`flask`** command won't be able to find it. Here an example `exampleapp.py` file that creates such an application:

```
from yourapplication import create_app
app = create_app('/path/to/config.cfg')
```

It can then be used with the **`flask`** command:

```
export FLASK_APP=exampleapp
flask run
```

### Factory Improvements

The factory function from above is not very clever so far, you can improve it. The following changes are straightforward and possible:

1. make it possible to pass in configuration values for unittests so that you don't have to create config files on the filesystem
2. call a function from a blueprint when the application is setting up so that you have a place to modify attributes of the application (like hooking in before / after request handlers etc.)
3. Add in WSGI middlewares when the application is creating if necessary.

## Application Dispatching

Application dispatching is the process of combining multiple Flask applications on the WSGI level. You can combine not only Flask applications but any WSGI application. This would allow you to run a Django and a Flask application in the same interpreter side by side if you want. The usefulness of this depends on how the applications work internally.

The fundamental difference from the *module approach* is that in this case you are running the same or different Flask applications that are entirely isolated from each other. They run different configurations and are dispatched on the WSGI level.

### Working with this Document

Each of the techniques and examples below results in an `application` object that can be run with any WSGI server. For production, see *Deployment Options*. For development, Werkzeug provides a builtin server for development available at `werkzeug.serving.run_simple()`:

```
from werkzeug.serving import run_simple
run_simple('localhost', 5000, application, use_reloader=True)
```

Note that `run_simple` is not intended for use in production. Use a *full-blown WSGI server*.

In order to use the interactive debugger, debugging must be enabled both on the application and the simple server. Here is the "hello world" example with debugging and `run_simple`:

```
from flask import Flask
from werkzeug.serving import run_simple

app = Flask(__name__)
app.debug = True

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    run_simple('localhost', 5000, app,
               use_reloader=True, use_debugger=True, use_evalex=True)
```

## Combining Applications

If you have entirely separated applications and you want them to work next to each other in the same Python interpreter process you can take advantage of the `werkzeug.wsgi.DispatcherMiddleware`. The idea here is that each Flask application is a valid WSGI application and they are combined by the dispatcher middleware into a larger one that is dispatched based on prefix.

For example you could have your main application run on `/` and your backend interface on `/backend`:

```
from werkzeug.wsgi import DispatcherMiddleware
from frontend_app import application as frontend
from backend_app import application as backend

application = DispatcherMiddleware(frontend, {
    '/backend':     backend
})
```

## Dispatch by Subdomain

Sometimes you might want to use multiple instances of the same application with different configurations. Assuming the application is created inside a function and you can call that function to instantiate it, that is really easy to implement. In order to

develop your application to support creating new instances in functions have a look at the *Application Factories* pattern.

A very common example would be creating applications per subdomain. For instance you configure your webserver to dispatch all requests for all subdomains to your application and you then use the subdomain information to create user-specific instances. Once you have your server set up to listen on all subdomains you can use a very simple WSGI application to do the dynamic application creation.

The perfect level for abstraction in that regard is the WSGI layer. You write your own WSGI application that looks at the request that comes and delegates it to your Flask application. If that application does not exist yet, it is dynamically created and remembered:

```
from threading import Lock

class SubdomainDispatcher(object):

    def __init__(self, domain, create_app):
        self.domain = domain
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, host):
        host = host.split(':')[0]
        assert host.endswith(self.domain), 'Configuration error'
        subdomain = host[:-len(self.domain)].rstrip('.')
        with self.lock:
            app = self.instances.get(subdomain)
            if app is None:
                app = self.create_app(subdomain)
                self.instances[subdomain] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(environ['HTTP_HOST'])
        return app(environ, start_response)
```

This dispatcher can then be used like this:

```
from myapplication import create_app, get_user_for_subdomain
from werkzeug.exceptions import NotFound

def make_app(subdomain):
    user = get_user_for_subdomain(subdomain)
    if user is None:
        # if there is no user for that subdomain we still have
        # to return a WSGI application that handles that request.
        # We can then just return the NotFound() exception as
        # application which will render a default 404 page.
        # You might also redirect the user to the main page then
```

```
        return NotFound()

    # otherwise create the application for the specific user
    return create_app(user)

application = SubdomainDispatcher('example.com', make_app)
```

## Dispatch by Path

Dispatching by a path on the URL is very similar. Instead of looking at the `Host` header to figure out the subdomain one simply looks at the request path up to the first slash:

```
from threading import Lock
from werkzeug.wsgi import pop_path_info, peek_path_info

class PathDispatcher(object):

    def __init__(self, default_app, create_app):
        self.default_app = default_app
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, prefix):
        with self.lock:
            app = self.instances.get(prefix)
            if app is None:
                app = self.create_app(prefix)
                if app is not None:
                    self.instances[prefix] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(peek_path_info(environ))
        if app is not None:
            pop_path_info(environ)
        else:
            app = self.default_app
        return app(environ, start_response)
```

The big difference between this and the subdomain one is that this one falls back to another application if the creator function returns `None`:

```
from myapplication import create_app, default_app, get_user_for_prefix

def make_app(prefix):
    user = get_user_for_prefix(prefix)
    if user is not None:
        return create_app(user)
```

```
application = PathDispatcher(default_app, make_app)
```

# Implementing API Exceptions

It's very common to implement RESTful APIs on top of Flask. One of the first things that developers run into is the realization that the builtin exceptions are not expressive enough for APIs and that the content type of `text/html` they are emitting is not very useful for API consumers.

The better solution than using `abort` to signal errors for invalid API usage is to implement your own exception type and install an error handler for it that produces the errors in the format the user is expecting.

## Simple Exception Class

The basic idea is to introduce a new exception that can take a proper human readable message, a status code for the error and some optional payload to give more context for the error.

This is a simple example:

```
from flask import jsonify

class InvalidUsage(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, payload=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv = dict(self.payload or ())
        rv['message'] = self.message
        return rv
```

A view can now raise that exception with an error message. Additionally some extra payload can be provided as a dictionary through the *payload* parameter.

## Registering an Error Handler

At that point views can raise that error, but it would immediately result in an internal server error. The reason for this is that there is no handler registered for this error

class. That however is easy to add:

```
@app.errorhandler(InvalidUsage)
def handle_invalid_usage(error):
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response
```

### Usage in Views

Here is how a view can use that functionality:

```
@app.route('/foo')
def get_foo():
    raise InvalidUsage('This view is gone', status_code=410)
```

## Using URL Processors

New in version 0.7.

Flask 0.7 introduces the concept of URL processors. The idea is that you might have a bunch of resources with common parts in the URL that you don't always explicitly want to provide. For instance you might have a bunch of URLs that have the language code in it but you don't want to have to handle it in every single function yourself.

URL processors are especially helpful when combined with blueprints. We will handle both application specific URL processors here as well as blueprint specifics.

### Internationalized Application URLs

Consider an application like this:

```
from flask import Flask, g

app = Flask(__name__)

@app.route('/<lang_code>/')
def index(lang_code):
    g.lang_code = lang_code
    ...

@app.route('/<lang_code>/about')
def about(lang_code):
    g.lang_code = lang_code
    ...
```

This is an awful lot of repetition as you have to handle the language code setting on the g object yourself in every single function. Sure, a decorator could be used to simplify this, but if you want to generate URLs from one function to another you would have to still provide the language code explicitly which can be annoying.

For the latter, this is where `url_defaults()` functions come in. They can automatically inject values into a call for `url_for()` automatically. The code below checks if the language code is not yet in the dictionary of URL values and if the endpoint wants a value named `'lang_code'`:

```
@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint_expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code
```

The method `is_endpoint_expecting()` of the URL map can be used to figure out if it would make sense to provide a language code for the given endpoint.

The reverse of that function are `url_value_preprocessor()`s. They are executed right after the request was matched and can execute code based on the URL values. The idea is that they pull information out of the values dictionary and put it somewhere else:

```
@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)
```

That way you no longer have to do the *lang_code* assignment to g in every function. You can further improve that by writing your own decorator that prefixes URLs with the language code, but the more beautiful solution is using a blueprint. Once the `'lang_code'` is popped from the values dictionary and it will no longer be forwarded to the view function reducing the code to this:

```
from flask import Flask, g

app = Flask(__name__)

@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint_expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code

@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)

@app.route('/<lang_code>/')
```

```
def index():
    ...

@app.route('/<lang_code>/about')
def about():
    ...
```

### Internationalized Blueprint URLs

Because blueprints can automatically prefix all URLs with a common string it's easy to automatically do that for every function. Furthermore blueprints can have per-blueprint URL processors which removes a whole lot of logic from the `url_defaults()` function because it no longer has to check if the URL is really interested in a `'lang_code'` parameter:

```
from flask import Blueprint, g

bp = Blueprint('frontend', __name__, url_prefix='/<lang_code>')

@bp.url_defaults
def add_language_code(endpoint, values):
    values.setdefault('lang_code', g.lang_code)

@bp.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code')

@bp.route('/')
def index():
    ...

@bp.route('/about')
def about():
    ...
```

## Deploying with Setuptools

Setuptools, is an extension library that is commonly used to distribute Python libraries and extensions. It extends distutils, a basic module installation system shipped with Python to also support various more complex constructs that make larger applications easier to distribute:

- **support for dependencies**: a library or application can declare a list of other libraries it depends on which will be installed automatically for you.
- **package registry**: setuptools registers your package with your Python installation. This makes it possible to query information provided by one package from

another package. The best known feature of this system is the entry point support which allows one package to declare an "entry point" that another package can hook into to extend the other package.

- **installation manager**: `pip` can install other libraries for you.

If you have Python 2 (>=2.7.9) or Python 3 (>=3.4) installed from python.org, you will already have pip and setuptools on your system. Otherwise, you will need to install them yourself.

Flask itself, and all the libraries you can find on PyPI are distributed with either setuptools or distutils.

In this case we assume your application is called `yourapplication.py` and you are not using a module, but a *package*. If you have not yet converted your application into a package, head over to the *Larger Applications* pattern to see how this can be done.

A working deployment with setuptools is the first step into more complex and more automated deployment scenarios. If you want to fully automate the process, also read the *Deploying with Fabric* chapter.

## Basic Setup Script

Because you have Flask installed, you have setuptools available on your system. Flask already depends upon setuptools.

Standard disclaimer applies: *you better use a virtualenv*.

Your setup code always goes into a file named `setup.py` next to your application. The name of the file is only convention, but because everybody will look for a file with that name, you better not change it.

A basic `setup.py` file for a Flask application looks like this:

```
from setuptools import setup

setup(
    name='Your Application',
    version='1.0',
    long_description=__doc__,
    packages=['yourapplication'],
    include_package_data=True,
    zip_safe=False,
    install_requires=['Flask']
)
```

Please keep in mind that you have to list subpackages explicitly. If you want setuptools to lookup the packages for you automatically, you can use the `find_packages` function:

```
from setuptools import setup, find_packages

setup(
```

```
    ...
    packages=find_packages()
)
```

Most parameters to the `setup` function should be self explanatory, `include_package_data` and `zip_safe` might not be. `include_package_data` tells setuptools to look for a `MANIFEST.in` file and install all the entries that match as package data. We will use this to distribute the static files and templates along with the Python module (see *Distributing Resources*). The `zip_safe` flag can be used to force or prevent zip Archive creation. In general you probably don't want your packages to be installed as zip files because some tools do not support them and they make debugging a lot harder.

## Tagging Builds

It is useful to distinguish between release and development builds. Add a `setup.cfg` file to configure these options.

[egg_info] tag_build = .dev tag_date = 1

[aliases] release = egg_info -RDb ''

Running `python setup.py sdist` will create a development package with ".dev" and the current date appended: `flaskr-1.0.dev20160314.tar.gz`. Running `python setup.py release sdist` will create a release package with only the version: `flaskr-1.0.tar.gz`.

## Distributing Resources

If you try to install the package you just created, you will notice that folders like `static` or `templates` are not installed for you. The reason for this is that setuptools does not know which files to add for you. What you should do, is to create a `MANIFEST.in` file next to your `setup.py` file. This file lists all the files that should be added to your tarball:

```
recursive-include yourapplication/templates *
recursive-include yourapplication/static *
```

Don't forget that even if you enlist them in your `MANIFEST.in` file, they won't be installed for you unless you set the *include_package_data* parameter of the `setup` function to `True`!

## Declaring Dependencies

Dependencies are declared in the `install_requires` parameter as a list. Each item in that list is the name of a package that should be pulled from PyPI on installation. By

default it will always use the most recent version, but you can also provide minimum and maximum version requirements. Here some examples:

```
install_requires=[
    'Flask>=0.2',
    'SQLAlchemy>=0.6',
    'BrokenPackage>=0.7,<=1.0'
]
```

As mentioned earlier, dependencies are pulled from PyPI. What if you want to depend on a package that cannot be found on PyPI and won't be because it is an internal package you don't want to share with anyone? Just do it as if there was a PyPI entry and provide a list of alternative locations where setuptools should look for tarballs:

```
dependency_links=['http://example.com/yourfiles']
```

Make sure that page has a directory listing and the links on the page are pointing to the actual tarballs with their correct filenames as this is how setuptools will find the files. If you have an internal company server that contains the packages, provide the URL to that server.

### Installing / Developing

To install your application (ideally into a virtualenv) just run the `setup.py` script with the `install` parameter. It will install your application into the virtualenv's site-packages folder and also download and install all dependencies:

```
$ python setup.py install
```

If you are developing on the package and also want the requirements to be installed, you can use the `develop` command instead:

```
$ python setup.py develop
```

This has the advantage of just installing a link to the site-packages folder instead of copying the data over. You can then continue to work on the code without having to run `install` again after each change.

## Deploying with Fabric

Fabric is a tool for Python similar to Makefiles but with the ability to execute commands on a remote server. In combination with a properly set up Python package (*Larger Applications*) and a good concept for configurations (*Configuration Handling*) it is very easy to deploy Flask applications to external servers.

Before we get started, here a quick checklist of things we have to ensure upfront:

- Fabric 1.0 has to be installed locally. This tutorial assumes the latest version of Fabric.
- The application already has to be a package and requires a working `setup.py` file (*Deploying with Setuptools*).
- In the following example we are using *mod_wsgi* for the remote servers. You can of course use your own favourite server there, but for this example we chose Apache + *mod_wsgi* because it's very easy to setup and has a simple way to reload applications without root access.

## Creating the first Fabfile

A fabfile is what controls what Fabric executes. It is named `fabfile.py` and executed by the *fab* command. All the functions defined in that file will show up as *fab* subcommands. They are executed on one or more hosts. These hosts can be defined either in the fabfile or on the command line. In this case we will add them to the fabfile.

This is a basic first example that has the ability to upload the current source code to the server and install it into a pre-existing virtual environment:

```
from fabric.api import *

# the user to use for the remote commands
env.user = 'appuser'
# the servers where the commands are executed
env.hosts = ['server1.example.com', 'server2.example.com']

def pack():
    # build the package
    local('python setup.py sdist --formats=gztar', capture=False)

def deploy():
    # figure out the package name and version
    dist = local('python setup.py --fullname', capture=True).strip()
    filename = '%s.tar.gz' % dist

    # upload the package to the temporary folder on the server
    put('dist/%s' % filename, '/tmp/%s' % filename)

    # install the package in the application's virtualenv with pip
    run('/var/www/yourapplication/env/bin/pip install /tmp/%s' % filename)

    # remove the uploaded package
    run('rm -r /tmp/%s' % filename)

    # touch the .wsgi file to trigger a reload in mod_wsgi
    run('touch /var/www/yourapplication.wsgi')
```

## Running Fabfiles

Now how do you execute that fabfile? You use the *fab* command. To deploy the current version of the code on the remote server you would use this command:

```
$ fab pack deploy
```

However this requires that our server already has the `/var/www/yourapplication` folder created and `/var/www/yourapplication/env` to be a virtual environment. Furthermore are we not creating the configuration or `.wsgi` file on the server. So how do we bootstrap a new server into our infrastructure?

This now depends on the number of servers we want to set up. If we just have one application server (which the majority of applications will have), creating a command in the fabfile for this is overkill. But obviously you can do that. In that case you would probably call it *setup* or *bootstrap* and then pass the servername explicitly on the command line:

```
$ fab -H newserver.example.com bootstrap
```

To setup a new server you would roughly do these steps:

1. Create the directory structure in `/var/www`:

   ```
   $ mkdir /var/www/yourapplication
   $ cd /var/www/yourapplication
   $ virtualenv --distribute env
   ```

2. Upload a new `application.wsgi` file to the server and the configuration file for the application (eg: `application.cfg`)
3. Create a new Apache config for `yourapplication` and activate it. Make sure to activate watching for changes of the `.wsgi` file so that we can automatically reload the application by touching it. (See *mod_wsgi (Apache)* for more information)

So now the question is, where do the `application.wsgi` and `application.cfg` files come from?

## The WSGI File

The WSGI file has to import the application and also to set an environment variable so that the application knows where to look for the config. This is a short example that does exactly that:

```
import os
os.environ['YOURAPPLICATION_CONFIG'] = '/var/www/yourapplication/application.cfg'
from yourapplication import app
```

The application itself then has to initialize itself like this to look for the config at that environment variable:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_config')
app.config.from_envvar('YOURAPPLICATION_CONFIG')
```

This approach is explained in detail in the *Configuration Handling* section of the documentation.

## The Configuration File

Now as mentioned above, the application will find the correct configuration file by looking up the `YOURAPPLICATION_CONFIG` environment variable. So we have to put the configuration in a place where the application will able to find it. Configuration files have the unfriendly quality of being different on all computers, so you do not version them usually.

A popular approach is to store configuration files for different servers in a separate version control repository and check them out on all servers. Then symlink the file that is active for the server into the location where it's expected (eg: `/var/www/yourapplication`).

Either way, in our case here we only expect one or two servers and we can upload them ahead of time by hand.

## First Deployment

Now we can do our first deployment. We have set up the servers so that they have their virtual environments and activated apache configs. Now we can pack up the application and deploy it:

```
$ fab pack deploy
```

Fabric will now connect to all servers and run the commands as written down in the fabfile. First it will execute pack so that we have our tarball ready and then it will execute deploy and upload the source code to all servers and install it there. Thanks to the `setup.py` file we will automatically pull in the required libraries into our virtual environment.

## Next Steps

From that point onwards there is so much that can be done to make deployment actually fun:

- Create a *bootstrap* command that initializes new servers. It could initialize a new virtual environment, setup apache appropriately etc.
- Put configuration files into a separate version control repository and symlink the active configs into place.

- You could also put your application code into a repository and check out the latest version on the server and then install. That way you can also easily go back to older versions.
- hook in testing functionality so that you can deploy to an external server and run the test suite.

Working with Fabric is fun and you will notice that it's quite magical to type `fab deploy` and see your application being deployed automatically to one or more remote servers.

## Using SQLite 3 with Flask

In Flask you can easily implement the opening of database connections on demand and closing them when the context dies (usually at the end of the request).

Here is a simple example of how you can use SQLite 3 with Flask:

```
import sqlite3
from flask import g

DATABASE = '/path/to/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

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

Now, to use the database, the application must either have an active application context (which is always true if there is a request in flight) or create an application context itself. At that point the `get_db` function can be used to get the current database connection. Whenever the context is destroyed the database connection will be terminated.

Note: if you use Flask 0.9 or older you need to use `flask._app_ctx_stack.top` instead of `g` as the `flask.g` object was bound to the request and not application context.

Example:

```
@app.route('/')
def index():
    cur = get_db().cursor()
    ...
```

**Note:** Please keep in mind that the teardown request and appcontext functions are always executed, even if a before-request handler failed or was never executed. Because of this we have to make sure here that the database is there before we close it.

## Connect on Demand

The upside of this approach (connecting on first use) is that this will only open the connection if truly necessary. If you want to use this code outside a request context you can use it in a Python shell by opening the application context by hand:

```
with app.app_context():
    # now you can use get_db()
```

## Easy Querying

Now in each request handling function you can access *g.db* to get the current open database connection. To simplify working with SQLite, a row factory function is useful. It is executed for every result returned from the database to convert the result. For instance, in order to get dictionaries instead of tuples, this could be inserted into the `get_db` function we created above:

```
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

db.row_factory = make_dicts
```

This will make the sqlite3 module return dicts for this database connection, which are much nicer to deal with. Even more simply, we could place this in `get_db` instead:

```
db.row_factory = sqlite3.Row
```

This would use Row objects rather than dicts to return the results of queries. These are `namedtuple` s, so we can access them either by index or by key. For example, assuming we have a `sqlite3.Row` called `r` for the rows `id`, `FirstName`, `LastName`, and `MiddleInitial`:

```
>>> # You can get values based on the row's name
>>> r['FirstName']
John
>>> # Or, you can get them based on index
>>> r[1]
John
# Row objects are also iterable:
>>> for value in r:
...     print(value)
```

```
1
John
Doe
M
```

Additionally, it is a good idea to provide a query function that combines getting the cursor, executing and fetching the results:

```
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

This handy little function, in combination with a row factory, makes working with the database much more pleasant than it is by just using the raw cursor and connection objects.

Here is how you can use it:

```
for user in query_db('select * from users'):
    print user['username'], 'has the id', user['user_id']
```

Or if you just want a single result:

```
user = query_db('select * from users where username = ?',
                [the_username], one=True)
if user is None:
    print 'No such user'
else:
    print the_username, 'has the id', user['user_id']
```

To pass variable parts to the SQL statement, use a question mark in the statement and pass in the arguments as a list. Never directly add them to the SQL statement with string formatting because this makes it possible to attack the application using SQL Injections.

## Initial Schemas

Relational databases need schemas, so applications often ship a *schema.sql* file that creates the database. It's a good idea to provide a function that creates the database based on that schema. This function can do that for you:

```
def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

You can then create such a database from the Python shell:

```
>>> from yourapplication import init_db
>>> init_db()
```

## SQLAlchemy in Flask

Many people prefer SQLAlchemy for database access. In this case it's encouraged to use a package instead of a module for your flask application and drop the models into a separate module (*Larger Applications*). While that is not necessary, it makes a lot of sense.

There are four very common ways to use SQLAlchemy. I will outline each of them here:

### Flask-SQLAlchemy Extension

Because SQLAlchemy is a common database abstraction layer and object relational mapper that requires a little bit of configuration effort, there is a Flask extension that handles that for you. This is recommended if you want to get started quickly.

You can download Flask-SQLAlchemy from PyPI.

### Declarative

The declarative extension in SQLAlchemy is the most recent method of using SQLAlchemy. It allows you to define tables and models in one go, similar to how Django works. In addition to the following text I recommend the official documentation on the declarative extension.

Here's the example `database.py` module for your application:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    # import all modules here that might define models so that
    # they will be registered properly on the metadata.  Otherwise
    # you will have to import them first before calling init_db()
```

```
    import yourapplication.models
    Base.metadata.create_all(bind=engine)
```

To define your models, just subclass the *Base* class that was created by the code above. If you are wondering why we don't have to care about threads here (like we did in the SQLite3 example above with the g object): that's because SQLAlchemy does that for us already with the `scoped_session`.

To use SQLAlchemy in a declarative way with your application, you just have to put the following code into your application module. Flask will automatically remove database sessions at the end of the request or when the application shuts down:

```
from yourapplication.database import db_session

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

Here is an example model (put this into `models.py`, e.g.):

```
from sqlalchemy import Column, Integer, String
from yourapplication.database import Base

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True)
    email = Column(String(120), unique=True)

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

    def __repr__(self):
        return '<User %r>' % (self.name)
```

To create the database you can use the *init_db* function:

```
>>> from yourapplication.database import init_db
>>> init_db()
```

You can insert entries into the database like this:

```
>>> from yourapplication.database import db_session
>>> from yourapplication.models import User
>>> u = User('admin', 'admin@localhost')
>>> db_session.add(u)
>>> db_session.commit()
```

Querying is simple as well:

```
>>> User.query.all()
[<User u'admin'>]
>>> User.query.filter(User.name == 'admin').first()
<User u'admin'>
```

## Manual Object Relational Mapping

Manual object relational mapping has a few upsides and a few downsides versus the declarative approach from above. The main difference is that you define tables and classes separately and map them together. It's more flexible but a little more to type. In general it works like the declarative approach, so make sure to also split up your application into multiple modules in a package.

Here is an example `database.py` module for your application:

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
metadata = MetaData()
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
def init_db():
    metadata.create_all(bind=engine)
```

As in the declarative approach, you need to close the session after each request or application context shutdown. Put this into your application module:

```
from yourapplication.database import db_session

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

Here is an example table and model (put this into `models.py`):

```
from sqlalchemy import Table, Column, Integer, String
from sqlalchemy.orm import mapper
from yourapplication.database import metadata, db_session

class User(object):
    query = db_session.query_property()

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

    def __repr__(self):
```

```
        return '<User %r>' % (self.name)

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50), unique=True),
    Column('email', String(120), unique=True)
)
mapper(User, users)
```

Querying and inserting works exactly the same as in the example above.

## SQL Abstraction Layer

If you just want to use the database system (and SQL) abstraction layer you basically only need the engine:

```
from sqlalchemy import create_engine, MetaData, Table

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
metadata = MetaData(bind=engine)
```

Then you can either declare the tables in your code like in the examples above, or automatically load them:

```
from sqlalchemy import Table

users = Table('users', metadata, autoload=True)
```

To insert data you can use the *insert* method. We have to get a connection first so that we can use a transaction:

```
>>> con = engine.connect()
>>> con.execute(users.insert(), name='admin', email='admin@localhost')
```

SQLAlchemy will automatically commit for us.

To query your database, you use the engine directly or use a connection:

```
>>> users.select(users.c.id == 1).execute().first()
(1, u'admin', u'admin@localhost')
```

These results are also dict-like tuples:

```
>>> r = users.select(users.c.id == 1).execute().first()
>>> r['name']
u'admin'
```

You can also pass strings of SQL statements to the `execute()` method:

```
>>> engine.execute('select * from users where id = :1', [1]).first()
(1, u'admin', u'admin@localhost')
```

For more information about SQLAlchemy, head over to the website.

# Uploading Files

Ah yes, the good old problem of file uploads. The basic idea of file uploads is actually quite simple. It basically works like this:

1. A `<form>` tag is marked with `enctype=multipart/form-data` and an `<input type=file>` is placed in that form.
2. The application accesses the file from the `files` dictionary on the request object.
3. use the `save()` method of the file to save the file permanently somewhere on the filesystem.

## A Gentle Introduction

Let's start with a very basic application that uploads a file to a specific upload folder and displays a file to the user. Let's look at the bootstrapping code for our application:

```
import os
from flask import Flask, request, redirect, url_for
from werkzeug.utils import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

So first we need a couple of imports. Most should be straightforward, the `werkzeug.secure_filename()` is explained a little bit later. The `UPLOAD_FOLDER` is where we will store the uploaded files and the `ALLOWED_EXTENSIONS` is the set of allowed file extensions.

Why do we limit the extensions that are allowed? You probably don't want your users to be able to upload everything there if the server is directly sending out the data to the client. That way you can make sure that users are not able to upload HTML files that would cause XSS problems (see *Cross-Site Scripting (XSS)*). Also make sure to disallow `.php` files if the server executes them, but who has PHP installed on their server, right? :)

Next the functions that check if an extension is valid and that uploads the file and redirects the user to the URL for the uploaded file:

```
def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        file = request.files['file']
        # if user does not select file, browser also
        # submit a empty part without filename
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('uploaded_file',
                                    filename=filename))
    return '''
    <!doctype html>
    <title>Upload new File</title>
    <h1>Upload new File</h1>
    <form method=post enctype=multipart/form-data>
      <p><input type=file name=file>
         <input type=submit value=Upload>
    </form>
    '''
```

So what does that `secure_filename()` function actually do? Now the problem is that there is that principle called "never trust user input". This is also true for the filename of an uploaded file. All submitted form data can be forged, and filenames can be dangerous. For the moment just remember: always use that function to secure a filename before storing it directly on the filesystem.

---

**Information for the Pros**

So you're interested in what that `secure_filename()` function does and what the problem is if you're not using it? So just imagine someone would send the following information as *filename* to your application:

```
filename = "../../../../home/username/.bashrc"
```

Assuming the number of `../` is correct and you would join this with the `UPLOAD_FOLDER` the user might have the ability to modify a file on the server's filesystem he or she should not modify. This does require some knowledge about how the application looks like, but trust me, hackers are patient :)

---

Now let's look how that function works:

```
>>> secure_filename('../../../../home/username/.bashrc')
'home_username_.bashrc'
```

---

Now one last thing is missing: the serving of the uploaded files. In the `upload_file()` we redirect the user to `url_for('uploaded_file', filename=filename)`, that is, `/uploads/filename`. So we write the `uploaded_file()` function to return the file of that name. As of Flask 0.5 we can use a function that does that for us:

```
from flask import send_from_directory

@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename)
```

Alternatively you can register *uploaded_file* as *build_only* rule and use the `SharedDataMiddleware`. This also works with older versions of Flask:

```
from werkzeug import SharedDataMiddleware
app.add_url_rule('/uploads/<filename>', 'uploaded_file',
                 build_only=True)
app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/uploads':  app.config['UPLOAD_FOLDER']
})
```

If you now run the application everything should work as expected.

## Improving Uploads

New in version 0.6.

So how exactly does Flask handle uploads? Well it will store them in the webserver's memory if the files are reasonable small otherwise in a temporary location (as returned by `tempfile.gettempdir()`). But how do you specify the maximum file size after which an upload is aborted? By default Flask will happily accept file uploads to an unlimited amount of memory, but you can limit that by setting the `MAX_CONTENT_LENGTH` config key:

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

The code above will limited the maximum allowed payload to 16 megabytes. If a larger file is transmitted, Flask will raise an `RequestEntityTooLarge` exception.

This feature was added in Flask 0.6 but can be achieved in older versions as well by

subclassing the request object. For more information on that consult the Werkzeug documentation on file handling.

### Upload Progress Bars

A while ago many developers had the idea to read the incoming file in small chunks and store the upload progress in the database to be able to poll the progress with JavaScript from the client. Long story short: the client asks the server every 5 seconds how much it has transmitted already. Do you realize the irony? The client is asking for something it should already know.

### An Easier Solution

Now there are better solutions that work faster and are more reliable. There are JavaScript libraries like jQuery that have form plugins to ease the construction of progress bar.

Because the common pattern for file uploads exists almost unchanged in all applications dealing with uploads, there is also a Flask extension called Flask-Uploads that implements a full fledged upload mechanism with white and blacklisting of extensions and more.

## Caching

When your application runs slow, throw some caches in. Well, at least it's the easiest way to speed up things. What does a cache do? Say you have a function that takes some time to complete but the results would still be good enough if they were 5 minutes old. So then the idea is that you actually put the result of that calculation into a cache for some time.

Flask itself does not provide caching for you, but Werkzeug, one of the libraries it is based on, has some very basic cache support. It supports multiple cache backends, normally you want to use a memcached server.

### Setting up a Cache

You create a cache object once and keep it around, similar to how `Flask` objects are created. If you are using the development server you can create a `SimpleCache` object, that one is a simple cache that keeps the item stored in the memory of the Python interpreter:

```
from werkzeug.contrib.cache import SimpleCache
cache = SimpleCache()
```

If you want to use memcached, make sure to have one of the memcache modules supported (you get them from PyPI) and a memcached server running somewhere. This is how you connect to such an memcached server then:

```
from werkzeug.contrib.cache import MemcachedCache
cache = MemcachedCache(['127.0.0.1:11211'])
```

If you are using App Engine, you can connect to the App Engine memcache server easily:

```
from werkzeug.contrib.cache import GAEMemcachedCache
cache = GAEMemcachedCache()
```

### Using a Cache

Now how can one use such a cache? There are two very important operations: `get()` and `set()`. This is how to use them:

To get an item from the cache call `get()` with a string as key name. If something is in the cache, it is returned. Otherwise that function will return `None`:

```
rv = cache.get('my-item')
```

To add items to the cache, use the `set()` method instead. The first argument is the key and the second the value that should be set. Also a timeout can be provided after which the cache will automatically remove item.

Here a full example how this looks like normally:

```
def get_my_item():
    rv = cache.get('my-item')
    if rv is None:
        rv = calculate_value()
        cache.set('my-item', rv, timeout=5 * 60)
    return rv
```

## View Decorators

Python has a really interesting feature called function decorators. This allows some really neat things for web applications. Because each view in Flask is a function, decorators can be used to inject additional functionality to one or more functions. The `route()` decorator is the one you probably used already. But there are use cases for implementing your own decorator. For instance, imagine you have a view that should only be used by people that are logged in. If a user goes to the site and is not logged in, they should be redirected to the login page. This is a good example of a use case where a decorator is an excellent solution.

## Login Required Decorator

So let's implement such a decorator. A decorator is a function that wraps and replaces another function. Since the original function is replaced, you need to remember to copy the original function's information to the new function. Use `functools.wraps()` to handle this for you.

This example assumes that the login page is called `'login'` and that the current user is stored in `g.user` and is `None` if there is no-one logged in.

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

To use the decorator, apply it as innermost decorator to a view function. When applying further decorators, always remember that the `route()` decorator is the outermost.

```
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

---

**Note:** The `next` value will exist in `request.args` after a `GET` request for the login page. You'll have to pass it along when sending the `POST` request from the login form. You can do this with a hidden input tag, then retrieve it from `request.form` when logging the user in.

```
<input type="hidden" value="{{ request.args.get('next', '') }}"/>
```

---

## Caching Decorator

Imagine you have a view function that does an expensive calculation and because of that you would like to cache the generated results for a certain amount of time. A decorator would be nice for that. We're assuming you have set up a cache like mentioned in *Caching*.

Here is an example cache function. It generates the cache key from a specific prefix (actually a format string) and the current path of the request. Notice that we are using a function that first creates the decorator that then decorates the function. Sounds aw-

ful? Unfortunately it is a little bit more complex, but the code should still be straightforward to read.

The decorated function will then work as follows

1. get the unique cache key for the current request base on the current path.
2. get the value for that key from the cache. If the cache returned something we will return that value.
3. otherwise the original function is called and the return value is stored in the cache for the timeout provided (by default 5 minutes).

Here the code:

```
from functools import wraps
from flask import request

def cached(timeout=5 * 60, key='view/%s'):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            cache_key = key % request.path
            rv = cache.get(cache_key)
            if rv is not None:
                return rv
            rv = f(*args, **kwargs)
            cache.set(cache_key, rv, timeout=timeout)
            return rv
        return decorated_function
    return decorator
```

Notice that this assumes an instantiated *cache* object is available, see *Caching* for more information.

## Templating Decorator

A common pattern invented by the TurboGears guys a while back is a templating decorator. The idea of that decorator is that you return a dictionary with the values passed to the template from the view function and the template is automatically rendered. With that, the following three examples do exactly the same:

```
@app.route('/')
def index():
    return render_template('index.html', value=42)

@app.route('/')
@templated('index.html')
def index():
    return dict(value=42)

@app.route('/')
```

```
@templated()
def index():
    return dict(value=42)
```

As you can see, if no template name is provided it will use the endpoint of the URL map with dots converted to slashes + `'.html'`. Otherwise the provided template name is used. When the decorated function returns, the dictionary returned is passed to the template rendering function. If `None` is returned, an empty dictionary is assumed, if something else than a dictionary is returned we return it from the function unchanged. That way you can still use the redirect function or return simple strings.

Here is the code for that decorator:

```
from functools import wraps
from flask import request, render_template

def templated(template=None):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            template_name = template
            if template_name is None:
                template_name = request.endpoint \
                    .replace('.', '/') + '.html'
            ctx = f(*args, **kwargs)
            if ctx is None:
                ctx = {}
            elif not isinstance(ctx, dict):
                return ctx
            return render_template(template_name, **ctx)
        return decorated_function
    return decorator
```

## Endpoint Decorator

When you want to use the werkzeug routing system for more flexibility you need to map the endpoint as defined in the `Rule` to a view function. This is possible with this decorator. For example:

```
from flask import Flask
from werkzeug.routing import Rule

app = Flask(__name__)
app.url_map.add(Rule('/', endpoint='index'))

@app.endpoint('index')
def my_index():
    return "Hello world"
```

# Form Validation with WTForms

When you have to work with form data submitted by a browser view, code quickly becomes very hard to read. There are libraries out there designed to make this process easier to manage. One of them is WTForms which we will handle here. If you find yourself in the situation of having many forms, you might want to give it a try.

When you are working with WTForms you have to define your forms as classes first. I recommend breaking up the application into multiple modules (*Larger Applications*) for that and adding a separate module for the forms.

---

**Getting the most out of WTForms with an Extension**

The Flask-WTF extension expands on this pattern and adds a few little helpers that make working with forms and Flask more fun. You can get it from PyPI.

---

## The Forms

This is an example form for a typical registration page:

```
from wtforms import Form, BooleanField, StringField, PasswordField, validators

class RegistrationForm(Form):
    username = StringField('Username', [validators.Length(min=4, max=25)])
    email = StringField('Email Address', [validators.Length(min=6, max=35)])
    password = PasswordField('New Password', [
        validators.DataRequired(),
        validators.EqualTo('confirm', message='Passwords must match')
    ])
    confirm = PasswordField('Repeat Password')
    accept_tos = BooleanField('I accept the TOS', [validators.DataRequired()])
```

## In the View

In the view function, the usage of this form looks like this:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(form.username.data, form.email.data,
                    form.password.data)
        db_session.add(user)
        flash('Thanks for registering')
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

Notice we're implying that the view is using SQLAlchemy here (*SQLAlchemy in Flask*), but that's not a requirement, of course. Adapt the code as necessary.

Things to remember:

1. create the form from the request `form` value if the data is submitted via the HTTP `POST` method and `args` if the data is submitted as `GET`.
2. to validate the data, call the `validate()` method, which will return `True` if the data validates, `False` otherwise.
3. to access individual values from the form, access *form.<NAME>.data*.

## Forms in Templates

Now to the template side. When you pass the form to the templates, you can easily render them there. Look at the following example template to see how easy this is. WTForms does half the form generation for us already. To make it even nicer, we can write a macro that renders a field with label and a list of errors if there are any.

Here's an example `_formhelpers.html` template with such a macro:

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
{% endmacro %}
```

This macro accepts a couple of keyword arguments that are forwarded to WTForm's field function, which renders the field for us. The keyword arguments will be inserted as HTML attributes. So, for example, you can call `render_field(form.username, class='username')` to add a class to the input element. Note that WTForms returns standard Python unicode strings, so we have to tell Jinja2 that this data is already HTML-escaped with the `|safe` filter.

Here is the `register.html` template for the function we used above, which takes advantage of the `_formhelpers.html` template:

```
{% from "_formhelpers.html" import render_field %}
<form method=post>
  <dl>
    {{ render_field(form.username) }}
    {{ render_field(form.email) }}
    {{ render_field(form.password) }}
    {{ render_field(form.confirm) }}
```

```
    {{ render_field(form.accept_tos) }}
  </dl>
  <p><input type=submit value=Register>
</form>
```

For more information about WTForms, head over to the WTForms website.

## Template Inheritance

The most powerful part of Jinja is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

Sounds complicated but is very basic. It's easiest to understand it by starting with an example.

### Base Template

This template, which we'll call `layout.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content:

```
<!doctype html>
<html>
  <head>
    {% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
      &copy; Copyright 2010 by <a href="http://domain.invalid/">you</a>.
      {% endblock %}
    </div>
  </body>
</html>
```

In this example, the `{% block %}` tags define four blocks that child templates can fill in. All the *block* tag does is tell the template engine that a child template may override those portions of the template.

### Child Template

A child template might look like this:

```
{% extends "layout.html" %}
{% block title %}Index{% endblock %}
{% block head %}
  {{ super() }}
  <style type="text/css">
    .important { color: #336699; }
  </style>
{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome on my awesome homepage.
{% endblock %}
```

The `{% extends %}` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent. The extends tag must be the first tag in the template. To render the contents of a block defined in the parent template, use `{{ super() }}`.

## Message Flashing

Good applications and user interfaces are all about feedback. If the user does not get enough feedback they will probably end up hating the application. Flask provides a really simple way to give feedback to a user with the flashing system. The flashing system basically makes it possible to record a message at the end of a request and access it next request and only next request. This is usually combined with a layout template that does this. Note that browsers and sometimes web servers enforce a limit on cookie sizes. This means that flashing messages that are too large for session cookies causes message flashing to fail silently.

### Simple Flashing

So here is a full example:

```
from flask import Flask, flash, redirect, render_template, \
     request, url_for

app = Flask(__name__)
app.secret_key = 'some_secret'

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

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != 'admin' or \
                request.form['password'] != 'secret':
            error = 'Invalid credentials'
        else:
            flash('You were successfully logged in')
            return redirect(url_for('index'))
    return render_template('login.html', error=error)
```

And here is the `layout.html` template which does the magic:

```
<!doctype html>
<title>My Application</title>
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

Here is the `index.html` template which inherits from `layout.html`:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Overview</h1>
  <p>Do you want to <a href="{{ url_for('login') }}">log in?</a>
{% endblock %}
```

And here is the `login.html` template which also inherits from `layout.html`:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Login</h1>
  {% if error %}
    <p class=error><strong>Error:</strong> {{ error }}
  {% endif %}
  <form method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username value="{{
          request.form.username }}">
      <dt>Password:
      <dd><input type=password name=password>
```

```
    </dl>
    <p><input type=submit value=Login>
  </form>
{% endblock %}
```

## Flashing With Categories

New in version 0.3.

It is also possible to provide categories when flashing a message. The default category if nothing is provided is `'message'`. Alternative categories can be used to give the user better feedback. For example error messages could be displayed with a red background.

To flash a message with a different category, just use the second argument to the `flash()` function:

```
flash(u'Invalid password provided', 'error')
```

Inside the template you then have to tell the `get_flashed_messages()` function to also return the categories. The loop looks slightly different in that situation then:

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

This is just one example of how to render these flashed messages. One might also use the category to add a prefix such as `<strong>Error:</strong>` to the message.

## Filtering Flash Messages

New in version 0.9.

Optionally you can pass a list of categories which filters the results of `get_flashed_messages()`. This is useful if you wish to render each category in a separate block.

```
{% with errors = get_flashed_messages(category_filter=["error"]) %}
{% if errors %}
<div class="alert-message block-message error">
  <a class="close" href="#">×</a>
  <ul>
```

```
    {%- for msg in errors %}
    <li>{{ msg }}</li>
    {% endfor -%}
  </ul>
</div>
{% endif %}
{% endwith %}
```

# AJAX with jQuery

jQuery is a small JavaScript library commonly used to simplify working with the DOM and JavaScript in general. It is the perfect tool to make web applications more dynamic by exchanging JSON between server and client.

JSON itself is a very lightweight transport format, very similar to how Python primitives (numbers, strings, dicts and lists) look like which is widely supported and very easy to parse. It became popular a few years ago and quickly replaced XML as transport format in web applications.

## Loading jQuery

In order to use jQuery, you have to download it first and place it in the static folder of your application and then ensure it's loaded. Ideally you have a layout template that is used for all pages where you just have to add a script statement to the bottom of your <body> to load jQuery:

```
<script type=text/javascript src="{{
  url_for('static', filename='jquery.js') }}"></script>
```

Another method is using Google's AJAX Libraries API to load jQuery:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<script>window.jQuery || document.write('<script src="{{
  url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

In this case you have to put jQuery into your static folder as a fallback, but it will first try to load it directly from Google. This has the advantage that your website will probably load faster for users if they went to at least one other website before using the same jQuery version from Google because it will already be in the browser cache.

## Where is My Site?

Do you know where your application is? If you are developing the answer is quite simple: it's on localhost port something and directly on the root of that server. But what if you later decide to move your application to a different location? For example

to `http://example.com/myapp`? On the server side this never was a problem because we were using the handy `url_for()` function that could answer that question for us, but if we are using jQuery we should not hardcode the path to the application but make that dynamic, so how can we do that?

A simple method would be to add a script tag to our page that sets a global variable to the prefix to the root of the application. Something like this:

```
<script type=text/javascript>
  $SCRIPT_ROOT = {{ request.script_root|tojson|safe }};
</script>
```

The `|safe` is necessary in Flask before 0.10 so that Jinja does not escape the JSON encoded string with HTML rules. Usually this would be necessary, but we are inside a `script` block here where different rules apply.

---

**Information for Pros**

In HTML the `script` tag is declared `CDATA` which means that entities will not be parsed. Everything until `</script>` is handled as script. This also means that there must never be any `</` between the script tags. `|tojson` is kind enough to do the right thing here and escape slashes for you (`{{ "</script>"|tojson|safe }}` is rendered as `"<\/script>"`).

In Flask 0.10 it goes a step further and escapes all HTML tags with unicode escapes. This makes it possible for Flask to automatically mark the result as HTML safe.

---

## JSON View Functions

Now let's create a server side function that accepts two URL arguments of numbers which should be added together and then sent back to the application in a JSON object. This is a really ridiculous example and is something you usually would do on the client side alone, but a simple example that shows how you would use jQuery and Flask nonetheless:

```
from flask import Flask, jsonify, render_template, request
app = Flask(__name__)

@app.route('/_add_numbers')
def add_numbers():
    a = request.args.get('a', 0, type=int)
    b = request.args.get('b', 0, type=int)
    return jsonify(result=a + b)

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

As you can see I also added an *index* method here that renders a template. This template will load jQuery as above and have a little form we can add two numbers and a link to trigger the function on the server side.

Note that we are using the `get()` method here which will never fail. If the key is missing a default value (here `0`) is returned. Furthermore it can convert values to a specific type (like in our case *int*). This is especially handy for code that is triggered by a script (APIs, JavaScript etc.) because you don't need special error reporting in that case.

## The HTML

Your index.html template either has to extend a `layout.html` template with jQuery loaded and the *$SCRIPT_ROOT* variable set, or do that on the top. Here's the HTML code needed for our little application (`index.html`). Notice that we also drop the script directly into the HTML here. It is usually a better idea to have that in a separate script file:

```
<script type=text/javascript>
  $(function() {
    $('a#calculate').bind('click', function() {
      $.getJSON($SCRIPT_ROOT + '/_add_numbers', {
        a: $('input[name="a"]').val(),
        b: $('input[name="b"]').val()
      }, function(data) {
        $("#result").text(data.result);
      });
      return false;
    });
  });
</script>
<h1>jQuery Example</h1>
<p><input type=text size=5 name=a> +
   <input type=text size=5 name=b> =
   <span id=result>?</span>
<p><a href=# id=calculate>calculate server side</a>
```

I won't go into detail here about how jQuery works, just a very quick explanation of the little bit of code above:

1. `$(function() { ... })` specifies code that should run once the browser is done loading the basic parts of the page.
2. `$('selector')` selects an element and lets you operate on it.
3. `element.bind('event', func)` specifies a function that should run when the user clicked on the element. If that function returns *false*, the default behavior will not kick in (in this case, navigate to the # URL).
4. `$.getJSON(url, data, func)` sends a `GET` request to *url* and will send the contents of the *data* object as query parameters. Once the data arrived, it will call

the given function with the return value as argument. Note that we can use the *$SCRIPT_ROOT* variable here that we set earlier.

If you don't get the whole picture, download the sourcecode for this example from GitHub.

# Custom Error Pages

Flask comes with a handy `abort()` function that aborts a request with an HTTP error code early. It will also provide a plain black and white error page for you with a basic description, but nothing fancy.

Depending on the error code it is less or more likely for the user to actually see such an error.

## Common Error Codes

The following error codes are some that are often displayed to the user, even if the application behaves correctly:

***404 Not Found*** The good old "chap, you made a mistake typing that URL" message. So common that even novices to the internet know that 404 means: damn, the thing I was looking for is not there. It's a very good idea to make sure there is actually something useful on a 404 page, at least a link back to the index.

***403 Forbidden*** If you have some kind of access control on your website, you will have to send a 403 code for disallowed resources. So make sure the user is not lost when they try to access a forbidden resource.

***410 Gone*** Did you know that there the "404 Not Found" has a brother named "410 Gone"? Few people actually implement that, but the idea is that resources that previously existed and got deleted answer with 410 instead of 404. If you are not deleting documents permanently from the database but just mark them as deleted, do the user a favour and use the 410 code instead and display a message that what they were looking for was deleted for all eternity.

***500 Internal Server Error*** Usually happens on programming errors or if the server is overloaded. A terribly good idea is to have a nice page there, because your application *will* fail sooner or later (see also: *Application Errors*).

## Error Handlers

An error handler is a function, just like a view function, but it is called when an error happens and is passed that error. The error is most likely a `HTTPException`, but in one case it can be a different error: a handler for internal server errors will be passed other exception instances as well if they are uncaught.

An error handler is registered with the `errorhandler()` decorator and the error code of the exception. Keep in mind that Flask will *not* set the error code for you, so make sure to also provide the HTTP status code when returning a response.

Please note that if you add an error handler for "500 Internal Server Error", Flask will not trigger it if it's running in Debug mode.

Here an example implementation for a "404 Page Not Found" exception:

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404
```

An example template might be this:

```
{% extends "layout.html" %}
{% block title %}Page Not Found{% endblock %}
{% block body %}
  <h1>Page Not Found</h1>
  <p>What you were looking for is just not there.
  <p><a href="{{ url_for('index') }}">go somewhere nice</a>
{% endblock %}
```

# Lazily Loading Views

Flask is usually used with the decorators. Decorators are simple and you have the URL right next to the function that is called for that specific URL. However there is a downside to this approach: it means all your code that uses decorators has to be imported upfront or Flask will never actually find your function.

This can be a problem if your application has to import quick. It might have to do that on systems like Google's App Engine or other systems. So if you suddenly notice that your application outgrows this approach you can fall back to a centralized URL mapping.

The system that enables having a central URL map is the `add_url_rule()` function. Instead of using decorators, you have a file that sets up the application with all URLs.

## Converting to Centralized URL Map

Imagine the current application looks somewhat like this:

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

@app.route('/')
def index():
```

```
    pass

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

Then, with the centralized approach you would have one file with the views (`views.py`) but without any decorator:

```
def index():
    pass

def user(username):
    pass
```

And then a file that sets up an application which maps the functions to URLs:

```
from flask import Flask
from yourapplication import views
app = Flask(__name__)
app.add_url_rule('/', view_func=views.index)
app.add_url_rule('/user/<username>', view_func=views.user)
```

## Loading Late

So far we only split up the views and the routing, but the module is still loaded upfront. The trick is to actually load the view function as needed. This can be accomplished with a helper class that behaves just like a function but internally imports the real function on first use:

```
from werkzeug import import_string, cached_property

class LazyView(object):

    def __init__(self, import_name):
        self.__module__, self.__name__ = import_name.rsplit('.', 1)
        self.import_name = import_name

    @cached_property
    def view(self):
        return import_string(self.import_name)

    def __call__(self, *args, **kwargs):
        return self.view(*args, **kwargs)
```

What's important here is is that *__module__* and *__name__* are properly set. This is used by Flask internally to figure out how to name the URL rules in case you don't provide a name for the rule yourself.

Then you can define your central place to combine the views like this:

```
from flask import Flask
from yourapplication.helpers import LazyView
app = Flask(__name__)
app.add_url_rule('/',
                 view_func=LazyView('yourapplication.views.index'))
app.add_url_rule('/user/<username>',
                 view_func=LazyView('yourapplication.views.user'))
```

You can further optimize this in terms of amount of keystrokes needed to write this by having a function that calls into `add_url_rule()` by prefixing a string with the project name and a dot, and by wrapping *view_func* in a *LazyView* as needed.

```
def url(import_name, url_rules=[], **options):
    view = LazyView('yourapplication.' + import_name)
    for url_rule in url_rules:
        app.add_url_rule(url_rule, view_func=view, **options)

# add a single route to the index view
url('views.index', ['/'])

# add two routes to a single function endpoint
url_rules = ['/user/','/user/<username>']
url('views.user', url_rules)
```

One thing to keep in mind is that before and after request handlers have to be in a file that is imported upfront to work properly on the first request. The same goes for any kind of remaining decorator.

# MongoKit in Flask

Using a document database rather than a full DBMS gets more common these days. This pattern shows how to use MongoKit, a document mapper library, to integrate with MongoDB.

This pattern requires a running MongoDB server and the MongoKit library installed.

There are two very common ways to use MongoKit. I will outline each of them here:

## Declarative

The default behavior of MongoKit is the declarative one that is based on common ideas from Django or the SQLAlchemy declarative extension.

Here an example `app.py` module for your application:

```
from flask import Flask
from mongokit import Connection, Document
```

```
# configuration
MONGODB_HOST = 'localhost'
MONGODB_PORT = 27017

# create the little application object
app = Flask(__name__)
app.config.from_object(__name__)

# connect to the database
connection = Connection(app.config['MONGODB_HOST'],
                        app.config['MONGODB_PORT'])
```

To define your models, just subclass the *Document* class that is imported from MongoKit. If you've seen the SQLAlchemy pattern you may wonder why we do not have a session and even do not define a *init_db* function here. On the one hand, MongoKit does not have something like a session. This sometimes makes it more to type but also makes it blazingly fast. On the other hand, MongoDB is schemaless. This means you can modify the data structure from one insert query to the next without any problem. MongoKit is just schemaless too, but implements some validation to ensure data integrity.

Here is an example document (put this also into `app.py`, e.g.):

```
from mongokit import ValidationError

def max_length(length):
    def validate(value):
        if len(value) <= length:
            return True
        # must have %s in error format string to have mongokit place key in there
        raise ValidationError('%s must be at most {} characters long'.
↪format(length))
    return validate

class User(Document):
    structure = {
        'name': unicode,
        'email': unicode,
    }
    validators = {
        'name': max_length(50),
        'email': max_length(120)
    }
    use_dot_notation = True
    def __repr__(self):
        return '<User %r>' % (self.name)

# register the User document with our current connection
connection.register([User])
```

This example shows you how to define your schema (named structure), a validator for the maximum character length and uses a special MongoKit feature called *use_dot_notation*. Per default MongoKit behaves like a python dictionary but with *use_dot_notation* set to `True` you can use your documents like you use models in nearly any other ORM by using dots to separate between attributes.

You can insert entries into the database like this:

```
>>> from yourapplication.database import connection
>>> from yourapplication.models import User
>>> collection = connection['test'].users
>>> user = collection.User()
>>> user['name'] = u'admin'
>>> user['email'] = u'admin@localhost'
>>> user.save()
```

Note that MongoKit is kinda strict with used column types, you must not use a common *str* type for either *name* or *email* but unicode.

Querying is simple as well:

```
>>> list(collection.User.find())
[<User u'admin'>]
>>> collection.User.find_one({'name': u'admin'})
<User u'admin'>
```

## PyMongo Compatibility Layer

If you just want to use PyMongo, you can do that with MongoKit as well. You may use this process if you need the best performance to get. Note that this example does not show how to couple it with Flask, see the above MongoKit code for examples:

```
from MongoKit import Connection

connection = Connection()
```

To insert data you can use the *insert* method. We have to get a collection first, this is somewhat the same as a table in the SQL world.

```
>>> collection = connection['test'].users
>>> user = {'name': u'admin', 'email': u'admin@localhost'}
>>> collection.insert(user)
```

MongoKit will automatically commit for us.

To query your database, you use the collection directly:

```
>>> list(collection.find())
[{u'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u
↪'admin@localhost'}]
```

```
>>> collection.find_one({'name': u'admin'})
{u'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u
↪'admin@localhost'}
```

These results are also dict-like objects:

```
>>> r = collection.find_one({'name': u'admin'})
>>> r['email']
u'admin@localhost'
```

For more information about MongoKit, head over to the website.

## Adding a favicon

A "favicon" is an icon used by browsers for tabs and bookmarks. This helps to distinguish your website and to give it a unique brand.

A common question is how to add a favicon to a Flask application. First, of course, you need an icon. It should be 16 × 16 pixels and in the ICO file format. This is not a requirement but a de-facto standard supported by all relevant browsers. Put the icon in your static directory as `favicon.ico`.

Now, to get browsers to find your icon, the correct way is to add a link tag in your HTML. So, for example:

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

That's all you need for most browsers, however some really old ones do not support this standard. The old de-facto standard is to serve this file, with this name, at the website root. If your application is not mounted at the root path of the domain you either need to configure the web server to serve the icon at the root or if you can't do that you're out of luck. If however your application is the root you can simply route a redirect:

```
app.add_url_rule('/favicon.ico',
                 redirect_to=url_for('static', filename='favicon.ico'))
```

If you want to save the extra redirect request you can also write a view using `send_from_directory()`:

```
import os
from flask import send_from_directory

@app.route('/favicon.ico')
def favicon():
    return send_from_directory(os.path.join(app.root_path, 'static'),
                               'favicon.ico', mimetype='image/vnd.microsoft.icon')
```

We can leave out the explicit mimetype and it will be guessed, but we may as well specify it to avoid the extra guessing, as it will always be the same.

The above will serve the icon via your application and if possible it's better to configure your dedicated web server to serve it; refer to the web server's documentation.

### See also

- The Favicon article on Wikipedia

## Streaming Contents

Sometimes you want to send an enormous amount of data to the client, much more than you want to keep in memory. When you are generating the data on the fly though, how do you send that back to the client without the roundtrip to the filesystem?

The answer is by using generators and direct responses.

### Basic Usage

This is a basic view function that generates a lot of CSV data on the fly. The trick is to have an inner function that uses a generator to generate data and to then invoke that function and pass it to a response object:

```
from flask import Response

@app.route('/large.csv')
def generate_large_csv():
    def generate():
        for row in iter_all_rows():
            yield ','.join(row) + '\n'
    return Response(generate(), mimetype='text/csv')
```

Each `yield` expression is directly sent to the browser. Note though that some WSGI middlewares might break streaming, so be careful there in debug environments with profilers and other things you might have enabled.

### Streaming from Templates

The Jinja2 template engine also supports rendering templates piece by piece. This functionality is not directly exposed by Flask because it is quite uncommon, but you can easily do it yourself:

```
from flask import Response

def stream_template(template_name, **context):
```

```
    app.update_template_context(context)
    t = app.jinja_env.get_template(template_name)
    rv = t.stream(context)
    rv.enable_buffering(5)
    return rv

@app.route('/my-large-page.html')
def render_large_template():
    rows = iter_all_rows()
    return Response(stream_template('the_template.html', rows=rows))
```

The trick here is to get the template object from the Jinja2 environment on the application and to call `stream()` instead of `render()` which returns a stream object instead of a string. Since we're bypassing the Flask template render functions and using the template object itself we have to make sure to update the render context ourselves by calling `update_template_context()`. The template is then evaluated as the stream is iterated over. Since each time you do a yield the server will flush the content to the client you might want to buffer up a few items in the template which you can do with `rv.enable_buffering(size)`. `5` is a sane default.

### Streaming with Context

New in version 0.9.

Note that when you stream data, the request context is already gone the moment the function executes. Flask 0.9 provides you with a helper that can keep the request context around during the execution of the generator:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

Without the `stream_with_context()` function you would get a `RuntimeError` at that point.

## Deferred Request Callbacks

One of the design principles of Flask is that response objects are created and passed down a chain of potential callbacks that can modify them or replace them. When the request handling starts, there is no response object yet. It is created as necessary either by a view function or by some other component in the system.

But what happens if you want to modify the response at a point where the response does not exist yet? A common example for that would be a before-request function that wants to set a cookie on the response object.

One way is to avoid the situation. Very often that is possible. For instance you can try to move that logic into an after-request callback instead. Sometimes however moving that code there is just not a very pleasant experience or makes code look very awkward.

As an alternative possibility you can attach a bunch of callback functions to the g object and call them at the end of the request. This way you can defer code execution from anywhere in the application.

## The Decorator

The following decorator is the key. It registers a function on a list on the g object:

```
from flask import g

def after_this_request(f):
    if not hasattr(g, 'after_request_callbacks'):
        g.after_request_callbacks = []
    g.after_request_callbacks.append(f)
    return f
```

## Calling the Deferred

Now you can use the *after_this_request* decorator to mark a function to be called at the end of the request. But we still need to call them. For this the following function needs to be registered as `after_request()` callback:

```
@app.after_request
def call_after_request_callbacks(response):
    for callback in getattr(g, 'after_request_callbacks', ()):
        callback(response)
    return response
```

## A Practical Example

At any time during a request, we can register a function to be called at the end of the request. For example you can remember the current language of the user in a cookie in the before-request function:

```
from flask import request

@app.before_request
def detect_user_language():
```

```
    language = request.cookies.get('user_lang')
    if language is None:
        language = guess_language_from_request()
        @after_this_request
        def remember_language(response):
            response.set_cookie('user_lang', language)
    g.language = language
```

## Adding HTTP Method Overrides

Some HTTP proxies do not support arbitrary HTTP methods or newer HTTP methods (such as PATCH). In that case it's possible to "proxy" HTTP methods through another HTTP method in total violation of the protocol.

The way this works is by letting the client do an HTTP POST request and set the `X-HTTP-Method-Override` header and set the value to the intended HTTP method (such as `PATCH`).

This can easily be accomplished with an HTTP middleware:

```
class HTTPMethodOverrideMiddleware(object):
    allowed_methods = frozenset([
        'GET',
        'HEAD',
        'POST',
        'DELETE',
        'PUT',
        'PATCH',
        'OPTIONS'
    ])
    bodyless_methods = frozenset(['GET', 'HEAD', 'OPTIONS', 'DELETE'])

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        method = environ.get('HTTP_X_HTTP_METHOD_OVERRIDE', '').upper()
        if method in self.allowed_methods:
            method = method.encode('ascii', 'replace')
            environ['REQUEST_METHOD'] = method
        if method in self.bodyless_methods:
            environ['CONTENT_LENGTH'] = '0'
        return self.app(environ, start_response)
```

To use this with Flask this is all that is necessary:

```
from flask import Flask
```

```
app = Flask(__name__)
app.wsgi_app = HTTPMethodOverrideMiddleware(app.wsgi_app)
```

## Request Content Checksums

Various pieces of code can consume the request data and preprocess it. For instance JSON data ends up on the request object already read and processed, form data ends up there as well but goes through a different code path. This seems inconvenient when you want to calculate the checksum of the incoming request data. This is necessary sometimes for some APIs.

Fortunately this is however very simple to change by wrapping the input stream.

The following example calculates the SHA1 checksum of the incoming data as it gets read and stores it in the WSGI environment:

```
import hashlib

class ChecksumCalcStream(object):

    def __init__(self, stream):
        self._stream = stream
        self._hash = hashlib.sha1()

    def read(self, bytes):
        rv = self._stream.read(bytes)
        self._hash.update(rv)
        return rv

    def readline(self, size_hint):
        rv = self._stream.readline(size_hint)
        self._hash.update(rv)
        return rv

def generate_checksum(request):
    env = request.environ
    stream = ChecksumCalcStream(env['wsgi.input'])
    env['wsgi.input'] = stream
    return stream._hash
```

To use this, all you need to do is to hook the calculating stream in before the request starts consuming data. (Eg: be careful accessing `request.form` or anything of that nature. `before_request_handlers` for instance should be careful not to access it).

Example usage:

```
@app.route('/special-api', methods=['POST'])
def special_api():
    hash = generate_checksum(request)
```

```
    # Accessing this parses the input stream
    files = request.files
    # At this point the hash is fully constructed.
    checksum = hash.hexdigest()
    return 'Hash was: %s' % checksum
```

# Celery Based Background Tasks

Celery is a task queue for Python with batteries included. It used to have a Flask integration but it became unnecessary after some restructuring of the internals of Celery with Version 3. This guide fills in the blanks in how to properly use Celery with Flask but assumes that you generally already read the First Steps with Celery guide in the official Celery documentation.

## Installing Celery

Celery is on the Python Package Index (PyPI), so it can be installed with standard Python tools like **pip** or **easy_install**:

```
$ pip install celery
```

## Configuring Celery

The first thing you need is a Celery instance, this is called the celery application. It serves the same purpose as the `Flask` object in Flask, just for Celery. Since this instance is used as the entry-point for everything you want to do in Celery, like creating tasks and managing workers, it must be possible for other modules to import it.

For instance you can place this in a `tasks` module. While you can use Celery without any reconfiguration with Flask, it becomes a bit nicer by subclassing tasks and adding support for Flask's application contexts and hooking it up with the Flask configuration.

This is all that is necessary to properly integrate Celery with Flask:

```
from celery import Celery

def make_celery(app):
    celery = Celery(app.import_name, backend=app.config['CELERY_RESULT_BACKEND'],
                    broker=app.config['CELERY_BROKER_URL'])
    celery.conf.update(app.config)
    TaskBase = celery.Task
    class ContextTask(TaskBase):
        abstract = True
        def __call__(self, *args, **kwargs):
```

```
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)
    celery.Task = ContextTask
    return celery
```

The function creates a new Celery object, configures it with the broker from the application config, updates the rest of the Celery config from the Flask config and then creates a subclass of the task that wraps the task execution in an application context.

## Minimal Example

With what we have above this is the minimal example of using Celery with Flask:

```
from flask import Flask

flask_app = Flask(__name__)
flask_app.config.update(
    CELERY_BROKER_URL='redis://localhost:6379',
    CELERY_RESULT_BACKEND='redis://localhost:6379'
)
celery = make_celery(flask_app)


@celery.task()
def add_together(a, b):
    return a + b
```

This task can now be called in the background:

```
>>> result = add_together.delay(23, 42)
>>> result.wait()
65
```

## Running the Celery Worker

Now if you jumped in and already executed the above code you will be disappointed to learn that your `.wait()` will never actually return. That's because you also need to run celery. You can do that by running celery as a worker:

```
$ celery -A your_application.celery worker
```

The `your_application` string has to point to your application's package or module that creates the *celery* object.

## Subclassing Flask

The `Flask` class is designed for subclassing.

For example, you may want to override how request parameters are handled to preserve their order:

```
from flask import Flask, Request
from werkzeug.datastructures import ImmutableOrderedMultiDict
class MyRequest(Request):
    """Request subclass to override request parameter storage"""
    parameter_storage_class = ImmutableOrderedMultiDict
class MyFlask(Flask):
    """Flask subclass using the custom request class"""
    request_class = MyRequest
```

This is the recommended approach for overriding or augmenting Flask's internal functionality.

CHAPTER **21**

# Deployment Options

While lightweight and easy to use, **Flask's built-in server is not suitable for production** as it doesn't scale well and by default serves only one request at a time. Some of the options available for properly running Flask in production are documented here.

If you want to deploy your Flask application to a WSGI server not listed here, look up the server documentation about how to use a WSGI app with it. Just remember that your `Flask` application object is the actual WSGI application.

## Hosted options

- Deploying Flask on Heroku
- Deploying Flask on OpenShift
- Deploying Flask on Webfaction
- Deploying Flask on Google App Engine
- Deploying Flask on AWS Elastic Beanstalk
- Sharing your Localhost Server with Localtunnel
- Deploying on Azure (IIS)
- Deploying on PythonAnywhere

# Self-hosted options

## mod_wsgi (Apache)

If you are using the Apache webserver, consider using mod_wsgi.

---

**Watch Out**

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to mod_wsgi.

---

### Installing *mod_wsgi*

If you don't have *mod_wsgi* installed yet you have to either install it using a package manager or compile it yourself. The mod_wsgi installation instructions cover source installations on UNIX systems.

If you are using Ubuntu/Debian you can apt-get it and activate it as follows:

```
# apt-get install libapache2-mod-wsgi
```

If you are using a yum based distribution (Fedora, OpenSUSE, etc..) you can install it as follows:

```
# yum install mod_wsgi
```

On FreeBSD install *mod_wsgi* by compiling the *www/mod_wsgi* port or by using pkg_add:

```
# pkg install ap22-mod_wsgi2
```

If you are using pkgsrc you can install *mod_wsgi* by compiling the *www/ap2-wsgi* package.

If you encounter segfaulting child processes after the first apache reload you can safely ignore them. Just restart the server.

### Creating a *.wsgi* file

To run your application you need a `yourapplication.wsgi` file. This file contains the code *mod_wsgi* is executing on startup to get the application object. The object called *application* in that file is then used as application.

For most applications the following file should be sufficient:

```
from yourapplication import app as application
```

If you don't have a factory function for application creation but a singleton instance you can directly import that one as *application*.

Store that file somewhere that you will find it again (e.g.: `/var/www/yourapplication`) and make sure that *yourapplication* and all the libraries that are in use are on the python load path. If you don't want to install it system wide consider using a virtual python instance. Keep in mind that you will have to actually install your application into the virtualenv as well. Alternatively there is the option to just patch the path in the `.wsgi` file before the import:

```
import sys
sys.path.insert(0, '/path/to/the/application')
```

### Configuring Apache

The last thing you have to do is to create an Apache configuration file for your application. In this example we are telling *mod_wsgi* to execute the application under a different user for security reasons:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess yourapplication user=user1 group=group1 threads=5
    WSGIScriptAlias / /var/www/yourapplication/yourapplication.wsgi

    <Directory /var/www/yourapplication>
        WSGIProcessGroup yourapplication
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Note: WSGIDaemonProcess isn't implemented in Windows and Apache will refuse to run with the above configuration. On a Windows system, eliminate those lines:

```
<VirtualHost *>
        ServerName example.com
        WSGIScriptAlias / C:\yourdir\yourapp.wsgi
        <Directory C:\yourdir>
                Order deny,allow
                Allow from all
        </Directory>
</VirtualHost>
```

Note: There have been some changes in access control configuration for Apache 2.4.

Most notably, the syntax for directory permissions has changed from httpd 2.2

```
Order allow,deny
Allow from all
```

to httpd 2.4 syntax

```
Require all granted
```

For more information consult the mod_wsgi documentation.

### Troubleshooting

If your application does not run, follow this guide to troubleshoot:

**Problem: application does not run, errorlog shows SystemExit ignored** You have an `app.run()` call in your application file that is not guarded by an `if __name__ == '__main__':` condition. Either remove that `run()` call from the file and move it into a separate `run.py` file or put it into such an if block.

**Problem: application gives permission errors** Probably caused by your application running as the wrong user. Make sure the folders the application needs access to have the proper privileges set and the application runs as the correct user (`user` and `group` parameter to the *WSGIDaemonProcess* directive)

**Problem: application dies with an error on print** Keep in mind that mod_wsgi disallows doing anything with `sys.stdout` and `sys.stderr`. You can disable this protection from the config by setting the *WSGIRestrictStdout* to `off`:

```
WSGIRestrictStdout Off
```

Alternatively you can also replace the standard out in the .wsgi file with a different stream:

```
import sys
sys.stdout = sys.stderr
```

**Problem: accessing resources gives IO errors** Your application probably is a single .py file you symlinked into the site-packages folder. Please be aware that this does not work, instead you either have to put the folder into the pythonpath the file is stored in, or convert your application into a package.

The reason for this is that for non-installed packages, the module filename is used to locate the resources and for symlinks the wrong filename is picked up.

### Support for Automatic Reloading

To help deployment tools you can activate support for automatic reloading. Whenever something changes the `.wsgi` file, *mod_wsgi* will reload all the daemon processes for us.

For that, just add the following directive to your *Directory* section:

```
WSGIScriptReloading On
```

#### Working with Virtual Environments

Virtual environments have the advantage that they never install the required dependencies system wide so you have a better control over what is used where. If you want to use a virtual environment with mod_wsgi you have to modify your `.wsgi` file slightly.

Add the following lines to the top of your `.wsgi` file:

```
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

For Python 3 add the following lines to the top of your `.wsgi` file:

```
activate_this = '/path/to/env/bin/activate_this.py'
with open(activate_this) as file_:
    exec(file_.read(), dict(__file__=activate_this))
```

This sets up the load paths according to the settings of the virtual environment. Keep in mind that the path has to be absolute.

### Standalone WSGI Containers

There are popular servers written in Python that contain WSGI applications and serve HTTP. These servers stand alone when they run; you can proxy to them from your web server. Note the section on *Proxy Setups* if you run into issues.

#### Gunicorn

Gunicorn 'Green Unicorn' is a WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project. It supports both eventlet and greenlet. Running a Flask application on this server is quite simple:

```
gunicorn myproject:app
```

Gunicorn provides many command-line options – see `gunicorn -h`. For example, to run a Flask application with 4 worker processes (`-w 4`) binding to localhost port 4000 (`-b 127.0.0.1:4000`):

```
gunicorn -w 4 -b 127.0.0.1:4000 myproject:app
```

### Gevent

Gevent is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of libev event loop:

```
from gevent.wsgi import WSGIServer
from yourapplication import app

http_server = WSGIServer(('', 5000), app)
http_server.serve_forever()
```

### Twisted Web

Twisted Web is the web server shipped with Twisted, a mature, non-blocking event-driven networking library. Twisted Web comes with a standard WSGI container which can be controlled from the command line using the `twistd` utility:

```
twistd web --wsgi myproject.app
```

This example will run a Flask application called `app` from a module named `myproject`.

Twisted Web supports many flags and options, and the `twistd` utility does as well; see `twistd -h` and `twistd web -h` for more information. For example, to run a Twisted Web server in the foreground, on port 8080, with an application from `myproject`:

```
twistd -n web --port 8080 --wsgi myproject.app
```

### Proxy Setups

If you deploy your application using one of these servers behind an HTTP proxy you will need to rewrite a few headers in order for the application to work. The two problematic values in the WSGI environment usually are `REMOTE_ADDR` and `HTTP_HOST`. You can configure your httpd to pass these headers, or you can fix them in middleware. Werkzeug ships a fixer that will solve some common setups, but you might want to write your own WSGI middleware for specific setups.

Here's a simple nginx configuration which proxies to an application served on localhost at port 8000, setting appropriate headers:

```
server {
    listen 80;

    server_name _;

    access_log  /var/log/nginx/access.log;
    error_log  /var/log/nginx/error.log;

    location / {
```

```
        proxy_pass          http://127.0.0.1:8000/;
        proxy_redirect      off;

        proxy_set_header    Host                $host;
        proxy_set_header    X-Real-IP           $remote_addr;
        proxy_set_header    X-Forwarded-For     $proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Proto   $scheme;
    }
}
```

If your httpd is not providing these headers, the most common setup invokes the host being set from `X-Forwarded-Host` and the remote address from `X-Forwarded-For`:

```
from werkzeug.contrib.fixers import ProxyFix
app.wsgi_app = ProxyFix(app.wsgi_app)
```

---

**Trusting Headers**

Please keep in mind that it is a security issue to use such a middleware in a non-proxy setup because it will blindly trust the incoming headers which might be forged by malicious clients.

---

If you want to rewrite the headers from another header, you might want to use a fixer like this:

```
class CustomProxyFix(object):

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        host = environ.get('HTTP_X_FHOST', '')
        if host:
            environ['HTTP_HOST'] = host
        return self.app(environ, start_response)

app.wsgi_app = CustomProxyFix(app.wsgi_app)
```

## uWSGI

uWSGI is a deployment option on servers like nginx, lighttpd, and cherokee; see *FastCGI* and *Standalone WSGI Containers* for other options. To use your WSGI application with uWSGI protocol you will need a uWSGI server first. uWSGI is both a protocol and an application server; the application server can serve uWSGI, FastCGI, and HTTP protocols.

The most popular uWSGI server is uwsgi, which we will use for this guide. Make sure to have it installed to follow along.

**Watch Out**

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to uWSGI.

### Starting your app with uwsgi

*uwsgi* is designed to operate on WSGI callables found in python modules.

Given a flask application in myapp.py, use the following command:

```
$ uwsgi -s /tmp/yourapplication.sock --manage-script-name --mount /
↪yourapplication=myapp:app
```

The `--manage-script-name` will move the handling of `SCRIPT_NAME` to uwsgi, since its smarter about that. It is used together with the `--mount` directive which will make requests to `/yourapplication` be directed to `myapp:app`. If your application is accessible at root level, you can use a single `/` instead of `/yourapplication`. `myapp` refers to the name of the file of your flask application (without extension) or the module which provides `app`. `app` is the callable inside of your application (usually the line reads `app = Flask(__name__)`.

If you want to deploy your flask application inside of a virtual environment, you need to also add `--virtualenv /path/to/virtual/environment`. You might also need to add `--plugin python` or `--plugin python3` depending on which python version you use for your project.

### Configuring nginx

A basic flask nginx configuration looks like this:

```
location = /yourapplication { rewrite ^ /yourapplication/; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
  include uwsgi_params;
  uwsgi_pass unix:/tmp/yourapplication.sock;
}
```

This configuration binds the application to `/yourapplication`. If you want to have it in the URL root its a bit simpler:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include uwsgi_params;
    uwsgi_pass unix:/tmp/yourapplication.sock;
}
```

## FastCGI

FastCGI is a deployment option on servers like nginx, lighttpd, and cherokee; see *uWSGI* and *Standalone WSGI Containers* for other options. To use your WSGI application with any of them you will need a FastCGI server first. The most popular one is flup which we will use for this guide. Make sure to have it installed to follow along.

---

**Watch Out**

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to FastCGI.

---

### Creating a *.fcgi* file

First you need to create the FastCGI server file. Let's call it *yourapplication.fcgi*:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from yourapplication import app

if __name__ == '__main__':
    WSGIServer(app).run()
```

This is enough for Apache to work, however nginx and older versions of lighttpd need a socket to be explicitly passed to communicate with the FastCGI server. For that to work you need to pass the path to the socket to the `WSGIServer`:

```
WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()
```

The path has to be the exact same path you define in the server config.

Save the `yourapplication.fcgi` file somewhere you will find it again. It makes sense to have that in `/var/www/yourapplication` or something similar.

Make sure to set the executable bit on that file so that the servers can execute it:

```
# chmod +x /var/www/yourapplication/yourapplication.fcgi
```

### Configuring Apache

The example above is good enough for a basic Apache deployment but your *.fcgi* file will appear in your application URL e.g. `example.com/yourapplication.fcgi/news/`.

There are few ways to configure your application so that yourapplication.fcgi does not appear in the URL. A preferable way is to use the ScriptAlias and SetHandler configuration directives to route requests to the FastCGI server. The following example uses FastCgiServer to start 5 instances of the application which will handle all incoming requests:

```
LoadModule fastcgi_module /usr/lib64/httpd/modules/mod_fastcgi.so

FastCgiServer /var/www/html/yourapplication/app.fcgi -idle-timeout 300 -processes 5

<VirtualHost *>
    ServerName webapp1.mydomain.com
    DocumentRoot /var/www/html/yourapplication

    AddHandler fastcgi-script fcgi
    ScriptAlias / /var/www/html/yourapplication/app.fcgi/

    <Location />
        SetHandler fastcgi-script
    </Location>
</VirtualHost>
```

These processes will be managed by Apache. If you're using a standalone FastCGI server, you can use the FastCgiExternalServer directive instead. Note that in the following the path is not real, it's simply used as an identifier to other directives such as AliasMatch:

```
FastCgiServer /var/www/html/yourapplication -host 127.0.0.1:3000
```

If you cannot set ScriptAlias, for example on a shared web host, you can use WSGI middleware to remove yourapplication.fcgi from the URLs. Set .htaccess:

```
<IfModule mod_fcgid.c>
   AddHandler fcgid-script .fcgi
   <Files ~ (\.fcgi)>
       SetHandler fcgid-script
       Options +FollowSymLinks +ExecCGI
   </Files>
</IfModule>

<IfModule mod_rewrite.c>
   Options +FollowSymlinks
   RewriteEngine On
   RewriteBase /
   RewriteCond %{REQUEST_FILENAME} !-f
   RewriteRule ^(.*)$ yourapplication.fcgi/$1 [QSA,L]
</IfModule>
```

Set yourapplication.fcgi:

```
#!/usr/bin/python
#: optional path to your local python site-packages folder
import sys
sys.path.insert(0, '<your_local_path>/lib/python2.6/site-packages')

from flup.server.fcgi import WSGIServer
from yourapplication import app

class ScriptNameStripper(object):
   def __init__(self, app):
       self.app = app

   def __call__(self, environ, start_response):
       environ['SCRIPT_NAME'] = ''
       return self.app(environ, start_response)

app = ScriptNameStripper(app)

if __name__ == '__main__':
    WSGIServer(app).run()
```

### Configuring lighttpd

A basic FastCGI configuration for lighttpd looks like that:

```
fastcgi.server = ("/yourapplication.fcgi" =>
    ((
        "socket" => "/tmp/yourapplication-fcgi.sock",
        "bin-path" => "/var/www/yourapplication/yourapplication.fcgi",
        "check-local" => "disable",
        "max-procs" => 1
    ))
)

alias.url = (
    "/static/" => "/path/to/your/static"
)

url.rewrite-once = (
    "^(/static($|/.*))$" => "$1",
    "^(/.*)$" => "/yourapplication.fcgi$1"
)
```

Remember to enable the FastCGI, alias and rewrite modules. This configuration binds the application to `/yourapplication`. If you want the application to work in the URL root you have to work around a lighttpd bug with the `LighttpdCGIRootFix` middleware.

Make sure to apply it only if you are mounting the application the URL root. Also,

see the Lighty docs for more information on FastCGI and Python (note that explicitly passing a socket to run() is no longer necessary).

### Configuring nginx

Installing FastCGI applications on nginx is a bit different because by default no FastCGI parameters are forwarded.

A basic Flask FastCGI configuration for nginx looks like this:

```
location = /yourapplication { rewrite ^ /yourapplication/ last; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
    include fastcgi_params;
    fastcgi_split_path_info ^(/yourapplication)(.*)$;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

This configuration binds the application to `/yourapplication`. If you want to have it in the URL root it's a bit simpler because you don't have to figure out how to calculate `PATH_INFO` and `SCRIPT_NAME`:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include fastcgi_params;
    fastcgi_param PATH_INFO $fastcgi_script_name;
    fastcgi_param SCRIPT_NAME "";
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

### Running FastCGI Processes

Since nginx and others do not load FastCGI apps, you have to do it by yourself. Supervisor can manage FastCGI processes. You can look around for other FastCGI process managers or write a script to run your *.fcgi* file at boot, e.g. using a SysV `init.d` script. For a temporary solution, you can always run the `.fcgi` script inside GNU screen. See `man screen` for details, and note that this is a manual solution which does not persist across system restart:

```
$ screen
$ /var/www/yourapplication/yourapplication.fcgi
```

### Debugging

FastCGI deployments tend to be hard to debug on most web servers. Very often the only thing the server log tells you is something along the lines of "premature end of

headers". In order to debug the application the only thing that can really give you ideas why it breaks is switching to the correct user and executing the application by hand.

This example assumes your application is called *application.fcgi* and that your web server user is *www-data*:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcgi", line 4, in <module>
ImportError: No module named yourapplication
```

In this case the error seems to be "yourapplication" not being on the python path. Common problems are:

- Relative paths being used. Don't rely on the current working directory.
- The code depending on environment variables that are not set by the web server.
- Different python interpreters being used.

## CGI

If all other deployment methods do not work, CGI will work for sure. CGI is supported by all major servers but usually has a sub-optimal performance.

This is also the way you can use a Flask application on Google's App Engine, where execution happens in a CGI-like environment.

---

**Watch Out**

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to CGI / app engine.

With CGI, you will also have to make sure that your code does not contain any `print` statements, or that `sys.stdout` is overridden by something that doesn't write into the HTTP response.

---

### Creating a *.cgi* file

First you need to create the CGI application file. Let's call it `yourapplication.cgi`:

```
#!/usr/bin/python
from wsgiref.handlers import CGIHandler
from yourapplication import app
```

```
CGIHandler().run(app)
```

### Server Setup

Usually there are two ways to configure the server. Either just copy the `.cgi` into a `cgi-bin` (and use *mod_rewrite* or something similar to rewrite the URL) or let the server point to the file directly.

In Apache for example you can put something like this into the config:

```
ScriptAlias /app /path/to/the/application.cgi
```

On shared webhosting, though, you might not have access to your Apache config. In this case, a file called `.htaccess`, sitting in the public directory you want your app to be available, works too but the `ScriptAlias` directive won't work in that case:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f # Don't interfere with static files
RewriteRule ^(.*)$ /path/to/the/application.cgi/$1 [L]
```

For more information consult the documentation of your webserver.

CHAPTER **22**

# Becoming Big

Here are your options when growing your codebase or scaling your application.

## Read the Source.

Flask started in part to demonstrate how to build your own framework on top of existing well-used tools Werkzeug (WSGI) and Jinja (templating), and as it developed, it became useful to a wide audience. As you grow your codebase, don't just use Flask – understand it. Read the source. Flask's code is written to be read; its documentation is published so you can use its internal APIs. Flask sticks to documented APIs in upstream libraries, and documents its internal utilities so that you can find the hook points needed for your project.

## Hook. Extend.

The *API* docs are full of available overrides, hook points, and *Signals*. You can provide custom classes for things like the request and response objects. Dig deeper on the APIs you use, and look for the customizations which are available out of the box in a Flask release. Look for ways in which your project can be refactored into a collection of utilities and Flask extensions. Explore the many extensions in the community, and look for patterns to build your own extensions if you do not find the tools you need.

## Subclass.

The `Flask` class has many methods designed for subclassing. You can quickly add or customize behavior by subclassing `Flask` (see the linked method docs) and using that subclass wherever you instantiate an application class. This works well with *Application Factories*. See *Subclassing Flask* for an example.

## Wrap with middleware.

The *Application Dispatching* chapter shows in detail how to apply middleware. You can introduce WSGI middleware to wrap your Flask instances and introduce fixes and changes at the layer between your Flask application and your HTTP server. Werkzeug includes several middlewares.

## Fork.

If none of the above options work, fork Flask. The majority of code of Flask is within Werkzeug and Jinja2. These libraries do the majority of the work. Flask is just the paste that glues those together. For every project there is the point where the underlying framework gets in the way (due to assumptions the original developers had). This is natural because if this would not be the case, the framework would be a very complex system to begin with which causes a steep learning curve and a lot of user frustration.

This is not unique to Flask. Many people use patched and modified versions of their framework to counter shortcomings. This idea is also reflected in the license of Flask. You don't have to contribute any changes back if you decide to modify the framework.

The downside of forking is of course that Flask extensions will most likely break because the new framework has a different import name. Furthermore integrating upstream changes can be a complex process, depending on the number of changes. Because of that, forking should be the very last resort.

## Scale like a pro.

For many web applications the complexity of the code is less an issue than the scaling for the number of users or data entries expected. Flask by itself is only limited in terms of scaling by your application code, the data store you want to use and the Python implementation and webserver you are running on.

Scaling well means for example that if you double the amount of servers you get about twice the performance. Scaling bad means that if you add a new server the application won't perform any better or would not even support a second server.

There is only one limiting factor regarding scaling in Flask which are the context local proxies. They depend on context which in Flask is defined as being either a thread, process or greenlet. If your server uses some kind of concurrency that is not based on threads or greenlets, Flask will no longer be able to support these global proxies. However the majority of servers are using either threads, greenlets or separate processes to achieve concurrency which are all methods well supported by the underlying Werkzeug library.

## Discuss with the community.

The Flask developers keep the framework accessible to users with codebases big and small. If you find an obstacle in your way, caused by Flask, don't hesitate to contact the developers on the mailinglist or IRC channel. The best way for the Flask and Flask extension developers to improve the tools for larger applications is getting feedback from users.

Part II

# API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

CHAPTER **23**

# API

This part of the documentation covers all the interfaces of Flask. For parts where Flask depends on external libraries, we document the most important right here and provide links to the canonical documentation.

## Application Object

**class** flask.**Flask**(*import_name*, *static_path=None*, *static_url_path=None*, *static_folder='static'*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=None*)

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see `open_resource()`.

Usually you create a `Flask` instance in your main module or in the `__init__.py` file of your package like this:

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

**About the First Parameter**

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, *__name__* is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in `yourapplication/app.py` you should create it with one of the two versions below:

```
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with *__name__*, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

New in version 0.7: The *static_url_path*, *static_folder*, and *template_folder* parameters were added.

New in version 0.8: The *instance_path* and *instance_relative_config* parameters were added.

New in version 0.11: The *root_path* parameter was added.

**Parameters**

- **`import_name`** – the name of the application package
- **`static_url_path`** – can be used to specify a different path for the static files on the web. Defaults to the name of the *static_folder* folder.
- **`static_folder`** – the folder with static files that should be served at *static_url_path*. Defaults to the `'static'` folder in the root path of the application.
- **`template_folder`** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.
- **`instance_path`** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.

- **instance_relative_config** – if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.
- **root_path** – Flask by default will automatically calculate the path to the root of the application. In certain situations this cannot be achieved (for instance if the package is a Python 3 namespace package) and needs to be manually defined.

**add_template_filter**(*f*, *name=None*)
Register a custom template filter. Works exactly like the `template_filter()` decorator.

**Parameters** **name** – the optional name of the filter, otherwise the function name will be used.

**add_template_global**(*f*, *name=None*)
Register a custom template global function. Works exactly like the `template_global()` decorator.

New in version 0.10.

**Parameters** **name** – the optional name of the global function, otherwise the function name will be used.

**add_template_test**(*f*, *name=None*)
Register a custom template test. Works exactly like the `template_test()` decorator.

New in version 0.10.

**Parameters** **name** – the optional name of the test, otherwise the function name will be used.

**add_url_rule**(*rule*, *endpoint=None*, *view_func=None*, ***options*)
Connects a URL rule. Works exactly like the `route()` decorator. If a view_func is provided it will be registered with the endpoint.

Basically this example:

```
@app.route('/')
def index():
    pass
```

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('/', 'index', index)
```

If the view_func is not provided you will need to connect the endpoint to a view function like so:

```
app.view_functions['index'] = index
```

Internally `route()` invokes `add_url_rule()` so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to *URL Route Registrations*.

Changed in version 0.2: *view_func* parameter added.

Changed in version 0.6: `OPTIONS` is added automatically as method.

**Parameters**

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **view_func** – the function to call when serving a request to the provided endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

**after_request**(*f*)

Register a function to be run after each request.

Your function must take one parameter, an instance of `response_class` and return a new response object or the same (see `process_response()`).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

**after_request_funcs = None**

A dictionary with lists of functions that should be called after each request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to close database connections. To register a function here, use the `after_request()` decorator.

**app_context**()

Binds the application only. For as long as the application is bound to the current context the `flask.current_app` points to that application. An application context is automatically created when a request context is pushed if necessary.

Example usage:

```
with app.app_context():
    ...
```

New in version 0.9.

**app_ctx_globals_class**

The class that is used for the g instance.

Example use cases for a custom class:

1.Store arbitrary attributes on flask.g.

2.Add a property for lazy per-request database connectors.

3.Return None instead of AttributeError on unexpected attributes.

4.Raise exception if an unexpected attr is set, a "controlled" flask.g.

In Flask 0.9 this property was called *request_globals_class* but it was changed in 0.10 to `app_ctx_globals_class` because the flask.g object is now application context scoped.

New in version 0.10.

alias of `_AppCtxGlobals`

**auto_find_instance_path**()
Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

New in version 0.8.

**before_first_request**(*f*)
Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

New in version 0.8.

**before_first_request_funcs = None**
A lists of functions that should be called at the beginning of the first request to this instance. To register a function here, use the `before_first_request()` decorator.

New in version 0.8.

**before_request**(*f*)
Registers a function to run before each request.

The function will be called without any arguments. If the function returns a non-None value, it's handled as if it was the return value from the view and further request handling is stopped.

**before_request_funcs = None**
A dictionary with lists of functions that should be called at the beginning of the request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to open database connections or getting hold of the currently logged in user. To register a function here, use the `before_request()` decorator.

**blueprints = None**
all the attached blueprints in a dictionary by name. Blueprints can be at-

tached multiple times so this dictionary does not tell you how often they got attached.

New in version 0.7.

**cli = None**
The click command line context for this application. Commands registered here show up in the **flask** command once the application has been discovered. The default commands are provided by Flask itself and can be overridden.

This is an instance of a `click.Group` object.

**config = None**
The configuration dictionary as `Config`. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

**config_class**
The class that is used for the `config` attribute of this app. Defaults to `Config`.

Example use cases for a custom class:

1. Default values for certain config options.
2. Access to config values through attributes in addition to keys.

New in version 0.11.

alias of `Config`

**context_processor**(*f*)
Registers a template context processor function.

**create_global_jinja_loader**()
Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the `jinja_loader()` function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

New in version 0.7.

**create_jinja_environment**()
Creates the Jinja2 environment based on `jinja_options` and `select_jinja_autoescape()`. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.

New in version 0.5.

Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

**create_url_adapter**(*request*)
Creates a URL adapter for the given request. The URL adapter is created at

a point where the request context is not yet set up so the request is passed explicitly.

New in version 0.6.

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

**debug**
The debug flag. Set this to `True` to enable debugging of the application. In debug mode the debugger will kick in when an unhandled exception occurs and the integrated server will automatically reload the application if changes in the code are detected.

This attribute can also be configured from the config with the `DEBUG` configuration key. Defaults to `False`.

**default_config = ImmutableDict({'JSON_AS_ASCII': True, 'USE_X_SENDFILE': False, 'SE**
Default configuration parameters.

**dispatch_request**()
Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call `make_response()`.

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

**do_teardown_appcontext**(*exc=<object object>*)
Called when an application context is popped. This works pretty much the same as `do_teardown_request()` but for the application context.

New in version 0.9.

**do_teardown_request**(*exc=<object object>*)
Called after the actual request dispatching and will call every as `teardown_request()` decorated function. This is not actually called by the `Flask` object itself but is always triggered when the request context is popped. That way we have a tighter control over certain resources under testing environments.

Changed in version 0.9: Added the *exc* argument. Previously this was always using the current exception information.

**endpoint**(*endpoint*)
A decorator to register a function as an endpoint. Example:

```
@app.endpoint('example.endpoint')
def example():
    return "example"
```

**Parameters** **endpoint** – the name of the endpoint

**error_handler_spec = None**

A dictionary of all registered error handlers. The key is `None` for error handlers active on the application, otherwise the key is the name of the blueprint. Each key points to another dictionary where the key is the status code of the http exception. The special key `None` points to a list of tuples where the first item is the class for the instance check and the second the error handler function.

To register a error handler, use the `errorhandler()` decorator.

**errorhandler**(*code_or_exception*)

A decorator that is used to register a function given an error code. Example:

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404
```

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

You can also register a function as error handler without using the `errorhandler()` decorator. The following example is equivalent to the one above:

```
def page_not_found(error):
    return 'This page does not exist', 404
app.error_handler_spec[None][404] = page_not_found
```

Setting error handlers via assignments to `error_handler_spec` however is discouraged as it requires fiddling with nested dictionaries and the special case for arbitrary exception types.

The first `None` refers to the active blueprint. If the error handler should be application wide `None` shall be used.

New in version 0.7: Use `register_error_handler()` instead of modifying `error_handler_spec` directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

**Parameters** **code_or_exception** – the code as integer for the handler, or an arbitrary exception

**extensions = None**

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things. For backwards compatibility extensions should register themselves like this:

```
if not hasattr(app, 'extensions'):
    app.extensions = {}
app.extensions['extensionname'] = SomeObject()
```

The key must match the name of the extension module. For example in case of a "Flask-Foo" extension in *flask_foo*, the key would be `'foo'`.

New in version 0.7.

**full_dispatch_request**()
Dispatches the request and on top of that performs request pre and post-processing as well as HTTP exception catching and error handling.

New in version 0.7.

**get_send_file_max_age**(*filename*)
Provides default cache_timeout for the `send_file()` functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`.

Static file functions such as `send_from_directory()` use this function, and `send_file()` calls this function on `current_app` when the given cache_timeout is `None`. If a cache_timeout is given in `send_file()`, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

New in version 0.9.

**got_first_request**
This attribute is set to `True` if the application started handling the first request.

New in version 0.8.

**handle_exception**(*e*)
Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

New in version 0.3.

**handle_http_exception**(*e*)
Handles an HTTP exception. By default this will invoke the registered error

handlers and fall back to returning the exception as response.

New in version 0.3.

**handle_url_build_error**(*error*, *endpoint*, *values*)
Handle `BuildError` on `url_for()`.

**handle_user_exception**(*e*)
This method is called whenever an exception occurs that should be handled. A special case are `HTTPExceptions` which are forwarded by this function to the `handle_http_exception()` method. This function will either return a response value or reraise the exception with the same traceback.

New in version 0.7.

**has_static_folder**
This is `True` if the package bound object's container has a folder for static files.

New in version 0.5.

**init_jinja_globals**()
Deprecated. Used to initialize the Jinja2 globals.

New in version 0.5.

Changed in version 0.7: This method is deprecated with 0.7. Override `create_jinja_environment()` instead.

**inject_url_defaults**(*endpoint*, *values*)
Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

New in version 0.7.

**instance_path = None**
Holds the path to the instance folder.

New in version 0.8.

**iter_blueprints**()
Iterates over all blueprints by the order they were registered.

New in version 0.11.

**jinja_env**
The Jinja2 environment used to load templates.

**jinja_environment**
The class that is used for the Jinja environment.

New in version 0.11.

alias of `Environment`

**jinja_loader**
The Jinja loader for this package bound object.

New in version 0.5.

**jinja_options = ImmutableDict({'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.with_']})**
Options that are passed directly to the Jinja2 environment.

**json_decoder**
The JSON decoder class to use. Defaults to `JSONDecoder`.

New in version 0.10.

alias of `JSONDecoder`

**json_encoder**
The JSON encoder class to use. Defaults to `JSONEncoder`.

New in version 0.10.

alias of `JSONEncoder`

**log_exception**(*exc_info*)
Logs an exception. This is called by `handle_exception()` if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the `logger`.

New in version 0.8.

**logger**
A `logging.Logger` object for this application. The default configuration is to log to stderr if the application is in debug mode. This logger can be used to (surprise) log messages. Here some examples:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

New in version 0.3.

**logger_name**
The name of the logger to use. By default the logger name is the package name passed to the constructor.

New in version 0.4.

**make_config**(*instance_relative=False*)
Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.

New in version 0.8.

**make_default_options_response**()
This method is called to create the default `OPTIONS` response. This can be changed through subclassing to change the default behavior of `OPTIONS` responses.

New in version 0.7.

**make_null_session**()
Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the session_interface.

New in version 0.7.

**make_response**(*rv*)
Converts the return value from a view function to a real response object that is an instance of response_class.

The following types are allowed for *rv*:

| response_class | the object is returned unchanged |
|---|---|
| str | a response object is created with the string as body |
| unicode | a response object is created with the string encoded to utf-8 as body |
| a WSGI function | the function is called as WSGI application and buffered as response object |
| tuple | A tuple in the form (response, status, headers) or (response, headers) where *response* is any of the types defined here, *status* is a string or an integer and *headers* is a list or a dictionary with header values. |

**Parameters** **rv** – the return value from the view function

Changed in version 0.9: Previously a tuple was interpreted as the arguments for the response object.

**make_shell_context**()
Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

New in version 0.11.

**name**
The name of the application. This is usually the import name with the difference that it's guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

New in version 0.8.

**open_instance_resource**(*resource*, *mode='rb'*)
Opens a resource from the application's instance folder (instance_path). Otherwise works like open_resource(). Instance resources can also be opened for writing.

**Parameters**

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.

- **mode** – resource file opening mode, default is 'rb'.

**open_resource**(*resource*, *mode='rb'*)

Opens a resource from the application's resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

**Parameters**

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

**open_session**(*request*)

Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the `secret_key` is set. Instead of overriding this method we recommend replacing the `session_interface`.

**Parameters** **request** – an instance of `request_class`.

**permanent_session_lifetime**

A `timedelta` which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the `PERMANENT_SESSION_LIFETIME` configuration key. Defaults to `timedelta(days=31)`

**preprocess_request**()

Called before the actual request dispatching and will call each `before_request()` decorated function, passing no arguments. If any of these functions returns a value, it's handled as if it was the return value from the view and further request handling is stopped.

This also triggers the `url_value_preprocessor()` functions before the actual `before_request()` functions are called.

**preserve_context_on_exception**
Returns the value of the PRESERVE_CONTEXT_ON_EXCEPTION configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

**process_response**(*response*)
Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the `after_request()` decorated functions.

Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

**Parameters** **response** – a `response_class` object.

**Returns** a new response object or the same, has to be an instance of `response_class`.

**propagate_exceptions**
Returns the value of the PROPAGATE_EXCEPTIONS configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

**register_blueprint**(*blueprint*, ***options*)
Registers a blueprint on the application.

New in version 0.7.

**register_error_handler**(*code_or_exception*, *f*)
Alternative error attach function to the `errorhandler()` decorator that is more straightforward to use for non decorator usage.

New in version 0.7.

**request_class**
The class that is used for request objects. See `Request` for more information.

alias of `Request`

**request_context**(*environ*)
Creates a `RequestContext` from the given environment and binds it to the current context. This must be used in combination with the `with` statement because the request is only bound to the current context for the duration of the `with` block.

Example usage:

```
with app.request_context(environ):
    do_something_with(request)
```

The object returned can also be used without the `with` statement which is useful for working in the shell. The example above is doing exactly the same as this code:

```
ctx = app.request_context(environ)
ctx.push()
try:
    do_something_with(request)
finally:
    ctx.pop()
```

Changed in version 0.3: Added support for non-with statement usage and `with` statement is now passed the ctx object.

**Parameters** **environ** – a WSGI environment

**response_class**
The class that is used for response objects. See `Response` for more information.

alias of `Response`

**route**(*rule*, ***options*)
A decorator that is used to register a view function for a given URL rule. This does the same thing as `add_url_rule()` but is intended for decorator usage:

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

For more information refer to *URL Route Registrations*.

**Parameters**

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

**run**(*host=None*, *port=None*, *debug=None*, ***options*)
Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see *Deployment Options* for WSGI server recommendations.

If the `debug` flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **`flask`** command line script's `run` support.

---

**Keep in Mind**

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke `run()` with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

---

Changed in version 0.10: The default port is now picked from the `SERVER_NAME` variable.

**Parameters**

- **`host`** – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'`.
- **`port`** – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
- **`debug`** – if given, enable or disable debug mode. See `debug`.
- **`options`** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.serving.run_simple()` for more information.

**`save_session`**(*session*, *response*)

Saves the session if it needs updates. For the default implementation, check `open_session()`. Instead of overriding this method we recommend replacing the `session_interface`.

**Parameters**

- **`session`** – the session to be saved (a `SecureCookie` object)
- **`response`** – an instance of `response_class`

**`secret_key`**

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the `SECRET_KEY` configuration key. Defaults to `None`.

**select_jinja_autoescape**(*filename*)
Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.

New in version 0.5.

**send_file_max_age_default**
A `timedelta` which is used as default cache_timeout for the `send_file()` functions. The default is 12 hours.

This attribute can also be configured from the config with the `SEND_FILE_MAX_AGE_DEFAULT` configuration key. This configuration variable can also be set with an integer value used as seconds. Defaults to `timedelta(hours=12)`

**send_static_file**(*filename*)
Function used internally to send static files from the static folder to the browser.

New in version 0.5.

**session_cookie_name**
The secure cookie uses this for the name of the session cookie.

This attribute can also be configured from the config with the `SESSION_COOKIE_NAME` configuration key. Defaults to `'session'`

**session_interface = <flask.sessions.SecureCookieSessionInterface object>**
the session interface to use. By default an instance of `SecureCookieSessionInterface` is used here.

New in version 0.8.

**shell_context_processor**(*f*)
Registers a shell context processor function.

New in version 0.11.

**shell_context_processors = None**
A list of shell context processor functions that should be run when a shell context is created.

New in version 0.11.

**should_ignore_error**(*error*)
This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

New in version 0.10.

**static_folder**
The absolute path to the configured static folder.

**teardown_appcontext**(*f*)
Registers a function to be called when the application context ends. These

functions are typically also called when the request context is popped.

Example:

```
ctx = app.app_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an exception it will be passed an error object.

The return values of teardown functions are ignored.

New in version 0.9.

**teardown_appcontext_funcs = None**

A list of functions that are called when the application context is destroyed. Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

New in version 0.9.

**teardown_request**(*f*)

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

```
ctx = app.test_request_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of a exception it will be passed an error object.

The return values of teardown functions are ignored.

**Debug Note**

In debug mode Flask will not tear down a request on an exception immediately. Instead it will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable.

**`teardown_request_funcs` = None**

A dictionary with lists of functions that are called after each request, even if an exception has occurred. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. These functions are not allowed to modify the request, and their return values are ignored. If an exception occurred while processing the request, it gets passed to each teardown_request function. To register a function here, use the `teardown_request()` decorator.

New in version 0.7.

**`template_context_processors` = None**

A dictionary with list of functions that are called without argument to populate the template context. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. Each returns a dictionary that the template context is updated with. To register a function here, use the `context_processor()` decorator.

**`template_filter`**(*name=None*)

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

```
@app.template_filter()
def reverse(s):
    return s[::-1]
```

**Parameters** **`name`** – the optional name of the filter, otherwise the function name will be used.

**`template_global`**(*name=None*)

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

New in version 0.10.

**Parameters** **`name`** – the optional name of the global function, otherwise the function name will be used.

**template_test**(*name=None*)

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

New in version 0.10.

**Parameters** **name** – the optional name of the test, otherwise the function name will be used.

**test_client**(*use_cookies=True*, ***kwargs*)

Creates a test client for this application. For information about unit testing head over to *Testing Flask Applications*.

Note that if you are testing for assertions or exceptions in your application code, you must set `app.testing = True` in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the `testing` attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a `with` block to defer the closing down of the context until the end of the `with` block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
    assert request.args['vodka'] == '42'
```

Additionally, you may pass optional keyword arguments that will then be passed to the application's `test_client_class` constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
    def __init__(self, *args, **kwargs):
        self._authentication = kwargs.pop("authentication")
        super(CustomClient,self).__init__( *args, **kwargs)

app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See `FlaskClient` for more information.

Changed in version 0.4: added support for `with` block usage for the client.

New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the `test_client_class` attribute.

Changed in version 0.11: Added ***kwargs* to support passing additional keyword arguments to the constructor of `test_client_class`.

**test_client_class = None**
the test client that is used with when *test_client* is used.

New in version 0.7.

**test_request_context**(**args*, ***kwargs*)
Creates a WSGI environment from the given values (see `werkzeug.test.EnvironBuilder` for more information, this function accepts the same arguments).

**testing**
The testing flag. Set this to `True` to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate unittest helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and PROPAGATE_EXCEPTIONS is not changed from the default it's implicitly enabled.

This attribute can also be configured from the config with the `TESTING` configuration key. Defaults to `False`.

**trap_http_exception**(*e*)
Checks if an HTTP exception should be trapped or not. By default this will return `False` for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to `True`. It also returns `True` if `TRAP_HTTP_EXCEPTIONS` is set to `True`.

This is called for all HTTP exceptions raised by a view function. If it returns `True` for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

New in version 0.8.

**update_template_context**(*context*)
Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

**Parameters** **context** – the context as a dictionary that is updated in place to add extra variables.

**url_build_error_handlers = None**
A list of functions that are called when `url_for()` raises a `BuildError`. Each function registered here is called with *error*, *endpoint* and *values*. If a function returns `None` or raises a `BuildError` the next function is tried.

New in version 0.9.

**url_default_functions = None**
A dictionary with lists of functions that can be used as URL value preprocessors. The key `None` here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary of URL values before they are used as the keyword arguments of the view function. For each function registered this one should also provide a `url_defaults()` function that adds the parameters automatically again that were removed that way.

New in version 0.7.

**url_defaults**(*f*)
Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

**url_map = None**
The `Map` for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

```
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def to_python(self, value):
        return value.split(',')
    def to_url(self, values):
        return ','.join(super(ListConverter, self).to_url(value)
                        for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter
```

**url_rule_class**
The rule object to use for URL rules created. This is used by `add_url_rule()`. Defaults to `werkzeug.routing.Rule`.

New in version 0.7.

alias of `Rule`

**url_value_preprocessor**(*f*)
Registers a function as URL value preprocessor for all view functions of the application. It's called before the view functions are called and can modify the url values provided.

**url_value_preprocessors = None**

A dictionary with lists of functions that can be used as URL value processor functions. Whenever a URL is built these functions are called to modify the dictionary of values in place. The key `None` here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary

New in version 0.7.

**use_x_sendfile**
Enable this if you want to use the X-Sendfile feature. Keep in mind that the server has to support this. This only affects files sent with the `send_file()` method.

New in version 0.2.

This attribute can also be configured from the config with the `USE_X_SENDFILE` configuration key. Defaults to `False`.

**view_functions = None**
A dictionary of all view functions registered. The keys will be function names which are also used to generate URLs and the values are the function objects themselves. To register a view function, use the `route()` decorator.

**wsgi_app**(*environ*, *start_response*)
The actual WSGI application. This is not implemented in *__call__* so that middlewares can be applied without losing a reference to the class. So instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it.

Changed in version 0.7: The behavior of the before and after request callbacks was changed under error conditions and a new callback was added that will always execute at the end of the request, independent on if an error occurred or not. See *Callbacks and Errors*.

**Parameters**

- **environ** – a WSGI environment
- **start_response** – a callable accepting a status code, a list of headers and an optional exception context to start the response

# Blueprint Objects

**class** flask.**Blueprint**(*name*, *import_name*, *static_folder=None*, *static_url_path=None*, *template_folder=None*, *url_prefix=None*, *subdomain=None*, *url_defaults=None*, *root_path=None*)

Represents a blueprint. A blueprint is an object that records functions that will be called with the `BlueprintSetupState` later to register functions or other things on the main application. See *Modular Applications with Blueprints* for more information.

New in version 0.7.

**add_app_template_filter**(*f*, *name=None*)

Register a custom template filter, available application wide. Like `Flask.add_template_filter()` but for a blueprint. Works exactly like the `app_template_filter()` decorator.

**Parameters** **name** – the optional name of the filter, otherwise the function name will be used.

**add_app_template_global**(*f*, *name=None*)

Register a custom template global, available application wide. Like `Flask.add_template_global()` but for a blueprint. Works exactly like the `app_template_global()` decorator.

New in version 0.10.

**Parameters** **name** – the optional name of the global, otherwise the function name will be used.

**add_app_template_test**(*f*, *name=None*)

Register a custom template test, available application wide. Like `Flask.add_template_test()` but for a blueprint. Works exactly like the `app_template_test()` decorator.

New in version 0.10.

**Parameters** **name** – the optional name of the test, otherwise the function name will be used.

**add_url_rule**(*rule*, *endpoint=None*, *view_func=None*, ***options*)

Like `Flask.add_url_rule()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

**after_app_request**(*f*)

Like `Flask.after_request()` but for a blueprint. Such a function is executed after each request, even if outside of the blueprint.

**after_request**(*f*)

Like `Flask.after_request()` but for a blueprint. This function is only executed after each request that is handled by a function of that blueprint.

**app_context_processor**(*f*)
Like `Flask.context_processor()` but for a blueprint. Such a function is executed each request, even if outside of the blueprint.

**app_errorhandler**(*code*)
Like `Flask.errorhandler()` but for a blueprint. This handler is used for all requests, even if outside of the blueprint.

**app_template_filter**(*name=None*)
Register a custom template filter, available application wide. Like `Flask.template_filter()` but for a blueprint.

**Parameters** **name** – the optional name of the filter, otherwise the function name will be used.

**app_template_global**(*name=None*)
Register a custom template global, available application wide. Like `Flask.template_global()` but for a blueprint.

New in version 0.10.

**Parameters** **name** – the optional name of the global, otherwise the function name will be used.

**app_template_test**(*name=None*)
Register a custom template test, available application wide. Like `Flask.template_test()` but for a blueprint.

New in version 0.10.

**Parameters** **name** – the optional name of the test, otherwise the function name will be used.

**app_url_defaults**(*f*)
Same as `url_defaults()` but application wide.

**app_url_value_preprocessor**(*f*)
Same as `url_value_preprocessor()` but application wide.

**before_app_first_request**(*f*)
Like `Flask.before_first_request()`. Such a function is executed before the first request to the application.

**before_app_request**(*f*)
Like `Flask.before_request()`. Such a function is executed before each request, even if outside of a blueprint.

**before_request**(*f*)
Like `Flask.before_request()` but for a blueprint. This function is only executed before each request that is handled by a function of that blueprint.

**context_processor**(*f*)
Like `Flask.context_processor()` but for a blueprint. This function is only executed for requests handled by a blueprint.

**endpoint**(*endpoint*)
Like `Flask.endpoint()` but for a blueprint. This does not prefix the endpoint with the blueprint name, this has to be done explicitly by the user of this method. If the endpoint is prefixed with a `.` it will be registered to the current blueprint, otherwise it's an application independent endpoint.

**errorhandler**(*code_or_exception*)
Registers an error handler that becomes active for this blueprint only. Please be aware that routing does not happen local to a blueprint so an error handler for 404 usually is not handled by a blueprint unless it is caused inside a view function. Another special case is the 500 internal server error which is always looked up from the application.

Otherwise works as the `errorhandler()` decorator of the `Flask` object.

**get_send_file_max_age**(*filename*)
Provides default cache_timeout for the `send_file()` functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`.

Static file functions such as `send_from_directory()` use this function, and `send_file()` calls this function on `current_app` when the given cache_timeout is `None`. If a cache_timeout is given in `send_file()`, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

New in version 0.9.

**has_static_folder**
This is `True` if the package bound object's container has a folder for static files.

New in version 0.5.

**jinja_loader**
The Jinja loader for this package bound object.

New in version 0.5.

**make_setup_state**(*app*, *options*, *first_registration=False*)
Creates an instance of `BlueprintSetupState()` object that is later passed to the register callback functions. Subclasses can override this to return a subclass of the setup state.

**open_resource**(*resource*, *mode='rb'*)
Opens a resource from the application's resource folder. To see how this

works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

**Parameters**

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

**record**(*func*)
Registers a function that is called when the blueprint is registered on the application. This function is called with the state as argument as returned by the `make_setup_state()` method.

**record_once**(*func*)
Works like `record()` but wraps the function in another function that will ensure the function is only called once. If the blueprint is registered a second time on the application, the function passed is not called.

**register**(*app*, *options*, *first_registration=False*)
Called by `Flask.register_blueprint()` to register a blueprint on the application. This can be overridden to customize the register behavior. Keyword arguments from `register_blueprint()` are directly forwarded to this method in the *options* dictionary.

**register_error_handler**(*code_or_exception*, *f*)
Non-decorator version of the `errorhandler()` error attach function, akin to the `register_error_handler()` application-wide function of the `Flask` object but for error handlers limited to this blueprint.

New in version 0.11.

**route**(*rule*, ***options*)
Like `Flask.route()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

**send_static_file**(*filename*)
Function used internally to send static files from the static folder to the browser.

New in version 0.5.

**static_folder**
The absolute path to the configured static folder.

**teardown_app_request**(*f*)
Like `Flask.teardown_request()` but for a blueprint. Such a function is executed when tearing down each request, even if outside of the blueprint.

**teardown_request**(*f*)
Like `Flask.teardown_request()` but for a blueprint. This function is only executed when tearing down requests handled by a function of that blueprint. Teardown request functions are executed when the request context is popped, even when no actual request was performed.

**url_defaults**(*f*)
Callback function for URL defaults for this blueprint. It's called with the endpoint and values and should update the values passed in place.

**url_value_preprocessor**(*f*)
Registers a function as URL value preprocessor for this blueprint. It's called before the view functions are called and can modify the url values provided.

## Incoming Request Data

**class** `flask.`**Request**(*environ*, *populate_request=True*, *shallow=False*)
The request object used by default in Flask. Remembers the matched endpoint and view arguments.

It is what ends up as `request`. If you want to replace the request object used you can subclass this and set `request_class` to your subclass.

The request object is a `Request` subclass and provides all of the attributes Werkzeug defines plus a few Flask specific ones.

**form**
A `MultiDict` with the parsed form data from `POST` or `PUT` requests. Please keep in mind that file uploads will not end up here, but instead in the `files` attribute.

**args**
A `MultiDict` with the parsed contents of the query string. (The part in the URL after the question mark).

**values**
A `CombinedMultiDict` with the contents of both `form` and `args`.

**cookies**
A `dict` with the contents of all cookies transmitted with the request.

**stream**
If the incoming form data was not encoded with a known mimetype the

data is stored unmodified in this stream for consumption. Most of the time it is a better idea to use `data` which will give you that data as a string. The stream only returns the data once.

**`headers`**
The incoming request headers as a dictionary like object.

**`data`**
Contains the incoming request data as string in case it came with a mimetype Flask does not handle.

**`files`**
A `MultiDict` with files uploaded as part of a `POST` or `PUT` request. Each file is stored as `FileStorage` object. It basically behaves like a standard file object you know from Python, with the difference that it also has a `save()` function that can store the file on the filesystem.

**`environ`**
The underlying WSGI environment.

**`method`**
The current request method (`POST`, `GET` etc.)

**`path`**

**`full_path`**

**`script_root`**

**`url`**

**`base_url`**

**`url_root`**
Provides different ways to look at the current IRI. Imagine your application is listening on the following application root:

```
http://www.example.com/myapplication
```

And a user requests the following URI:

```
http://www.example.com/myapplication/%CF%80/page.html?x=y
```

In this case the values of the above mentioned attributes would be the following:

| | |
|---|---|
| *path* | `u'/π/page.html'` |
| *full_path* | `u'/π/page.html?x=y'` |
| *script_root* | `u'/myapplication'` |
| *base_url* | `u'http://www.example.com/myapplication/π/page.html'` |
| *url* | `u'http://www.example.com/myapplication/π/page.html?x=y'` |
| *url_root* | `u'http://www.example.com/myapplication/'` |

**is_xhr**
: `True` if the request was triggered via a JavaScript *XMLHttpRequest*. This only works with libraries that support the `X-Requested-With` header and set it to *XMLHttpRequest*. Libraries that do that are prototype, jQuery and Mochikit and probably some more.

**blueprint**
: The name of the current blueprint

**endpoint**
: The endpoint that matched the request. This in combination with `view_args` can be used to reconstruct the same or a modified URL. If an exception happened when matching, this will be `None`.

**get_json**(*force=False*, *silent=False*, *cache=True*)
: Parses the incoming JSON request data and returns it. By default this function will return `None` if the mimetype is not `application/json` but this can be overridden by the `force` parameter. If parsing fails the `on_json_loading_failed()` method on the request object will be invoked.

  **Parameters**

  - **force** – if set to `True` the mimetype is ignored.
  - **silent** – if set to `True` this method will fail silently and return `None`.
  - **cache** – if set to `True` the parsed JSON data is remembered on the request.

**is_json**
: Indicates if this request is JSON or not. By default a request is considered to include JSON data if the mimetype is `application/json` or `application/*+json`.

  New in version 0.11.

**json**
: If the mimetype is `application/json` this will contain the parsed JSON data. Otherwise this will be `None`.

  The `get_json()` method should be used instead.

**max_content_length**
: Read-only view of the `MAX_CONTENT_LENGTH` config key.

**module**
: The name of the current module if the request was dispatched to an actual module. This is deprecated functionality, use blueprints instead.

**on_json_loading_failed**(*e*)
: Called if decoding of the JSON data failed. The return value of this method is used by `get_json()` when an error occurred. The default implementation just raises a `BadRequest` exception.

Changed in version 0.10: Removed buggy previous behavior of generating a random JSON response. If you want that behavior back you can trivially add it by subclassing.

New in version 0.8.

**routing_exception = None**
If matching the URL failed, this is the exception that will be raised / was raised as part of the request handling. This is usually a `NotFound` exception or something similar.

**url_rule = None**
The internal URL rule that matched the request. This can be useful to inspect which methods are allowed for the URL from a before/after handler (`request.url_rule.methods`) etc.

New in version 0.6.

**view_args = None**
A dict of view arguments that matched the request. If an exception happened when matching, this will be `None`.

**class** `flask.`**request**
To access incoming request data, you can use the global *request* object. Flask parses incoming request data for you and gives you access to it through that global object. Internally Flask makes sure that you always get the correct data for the active thread if you are in a multithreaded environment.

This is a proxy. See *Notes On Proxies* for more information.

The request object is an instance of a `Request` subclass and provides all of the attributes Werkzeug defines. This just shows a quick overview of the most important ones.

## Response Objects

**class** `flask.`**Response**(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)
The response object that is used by default in Flask. Works like the response object from Werkzeug but is set to have an HTML mimetype by default. Quite often you don't have to create this object yourself because `make_response()` will take care of that for you.

If you want to replace the response object used you can subclass this and set `response_class` to your subclass.

**headers**
A `Headers` object representing the response headers.

**status**
A string with a response status.

**status_code**
The response status as integer.

**data**
A descriptor that calls `get_data()` and `set_data()`. This should not be used and will eventually get deprecated.

**mimetype**
The mimetype (content type without charset etc.)

**set_cookie**(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*)
Sets a cookie. The parameters are the same as in the cookie *Morsel* object in the Python standard library but it accepts unicode data, too.

**Parameters**

- **key** – the key (name) of the cookie to be set.
- **value** – the value of the cookie.
- **max_age** – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client's browser session.
- **expires** – should be a *datetime* object or UNIX timestamp.
- **path** – limits the cookie to a given path, per default it will span the whole domain.
- **domain** – if you want to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domain `www.example.com`, `foo.example.com` etc. Otherwise, a cookie will only be readable by the domain that set it.
- **secure** – If *True*, the cookie will only be available via HTTPS
- **httponly** – disallow JavaScript to access the cookie. This is an extension to the cookie standard and probably not supported by all browsers.

## Sessions

If you have the `Flask.secret_key` set you can use sessions in Flask applications. A session basically makes it possible to remember information from one request to another. The way Flask does this is by using a signed cookie. So the user can look at the session contents, but not modify it unless they know the secret key, so make sure to set that to something complex and unguessable.

To access the current session you can use the `session` object:

**class** flask.**session**

The session object works pretty much like an ordinary dict, with the difference that it keeps track on modifications.

This is a proxy. See *Notes On Proxies* for more information.

The following attributes are interesting:

**new**

`True` if the session is new, `False` otherwise.

**modified**

`True` if the session object detected a modification. Be advised that modifications on mutable structures are not picked up automatically, in that situation you have to explicitly set the attribute to `True` yourself. Here an example:

```
# this change is not picked up because a mutable object (here
# a list) is changed.
session['objects'].append(42)
# so mark it as modified yourself
session.modified = True
```

**permanent**

If set to `True` the session lives for `permanent_session_lifetime` seconds. The default is 31 days. If set to `False` (which is the default) the session will be deleted when the user closes the browser.

## Session Interface

New in version 0.8.

The session interface provides a simple way to replace the session implementation that Flask is using.

**class** flask.sessions.**SessionInterface**

The basic interface you have to implement in order to replace the default session interface which uses werkzeug's securecookie implementation. The only methods you have to implement are `open_session()` and `save_session()`, the others have useful defaults which you don't need to change.

The session object returned by the `open_session()` method has to provide a dictionary like interface plus the properties and methods from the `SessionMixin`. We recommend just subclassing a dict and adding that mixin:

```
class Session(dict, SessionMixin):
    pass
```

If `open_session()` returns `None` Flask will call into `make_null_session()` to create a session that acts as replacement if the session support cannot work because some requirement is not fulfilled. The default `NullSession` class that is created will complain that the secret key was not set.

To replace the session interface on an application all you have to do is to assign `flask.Flask.session_interface`:

```
app = Flask(__name__)
app.session_interface = MySessionInterface()
```

New in version 0.8.

**get_cookie_domain**(*app*)
Helpful helper method that returns the cookie domain that should be used for the session cookie if session cookies are used.

**get_cookie_httponly**(*app*)
Returns True if the session cookie should be httponly. This currently just returns the value of the `SESSION_COOKIE_HTTPONLY` config var.

**get_cookie_path**(*app*)
Returns the path for which the cookie should be valid. The default implementation uses the value from the `SESSION_COOKIE_PATH` config var if it's set, and falls back to `APPLICATION_ROOT` or uses `/` if it's `None`.

**get_cookie_secure**(*app*)
Returns True if the cookie should be secure. This currently just returns the value of the `SESSION_COOKIE_SECURE` setting.

**get_expiration_time**(*app*, *session*)
A helper method that returns an expiration date for the session or `None` if the session is linked to the browser session. The default implementation returns now + the permanent session lifetime configured on the application.

**is_null_session**(*obj*)
Checks if a given object is a null session. Null sessions are not asked to be saved.

This checks if the object is an instance of `null_session_class` by default.

**make_null_session**(*app*)
Creates a null session which acts as a replacement object if the real session support could not be loaded due to a configuration error. This mainly aids the user experience because the job of the null session is to still support lookup without complaining but modifications are answered with a helpful error message of what failed.

This creates an instance of `null_session_class` by default.

**null_session_class**
`make_null_session()` will look here for the class that should be created when a null session is requested. Likewise the `is_null_session()` method will perform a typecheck against this type.

alias of `NullSession`

**open_session**(*app*, *request*)
This method has to be implemented and must either return `None` in case

the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on `SessionMixin`.

**pickle_based = False**
A flag that indicates if the session interface is pickle based. This can be used by Flask extensions to make a decision in regards to how to deal with the session object.

New in version 0.10.

**save_session**(*app*, *session*, *response*)
This is called for actual sessions returned by `open_session()` at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.

**should_set_cookie**(*app*, *session*)
Indicates whether a cookie should be set now or not. This is used by session backends to figure out if they should emit a set-cookie header or not. The default behavior is controlled by the `SESSION_REFRESH_EACH_REQUEST` config variable. If it's set to `False` then a cookie is only set if the session is modified, if set to `True` it's always set if the session is permanent.

This check is usually skipped if sessions get deleted.

New in version 0.11.

**class** `flask.sessions.`**SecureCookieSessionInterface**
The default session interface that stores sessions in signed cookies through the `itsdangerous` module.

**static digest_method**()
the hash function to use for the signature. The default is sha1

**key_derivation = 'hmac'**
the name of the itsdangerous supported key derivation. The default is hmac.

**salt = 'cookie-session'**
the salt that should be applied on top of the secret key for the signing of cookie based sessions.

**serializer = <flask.sessions.TaggedJSONSerializer object>**
A python serializer for the payload. The default is a compact JSON derived serializer with support for some extra Python types such as datetime objects or tuples.

**session_class**
alias of `SecureCookieSession`

**class** `flask.sessions.`**SecureCookieSession**(*initial=None*)
Base class for sessions based on signed cookies.

**class** `flask.sessions.`**NullSession**(*initial=None*)
Class used to generate nicer error messages if sessions are not available. Will still

allow read-only access to the empty session but fail on setting.

**class** flask.sessions.**SessionMixin**
Expands a basic dictionary with an accessors that are expected by Flask extensions and users for the session.

**modified = True**
for some backends this will always be True, but some backends will default this to false and detect changes in the dictionary for as long as changes do not happen on mutable structures in the session. The default mixin implementation just hardcodes True in.

**new = False**
some session backends can tell you if a session is new, but that is not necessarily guaranteed. Use with caution. The default mixin implementation just hardcodes False in.

**permanent**
this reflects the '_permanent' key in the dict.

flask.sessions.**session_json_serializer = <flask.sessions.TaggedJSONSerializer object>**
A customized JSON serializer that supports a few extra types that we take for granted when serializing (tuples, markup objects, datetime).

This object provides dumping and loading methods similar to simplejson but it also tags certain builtin Python objects that commonly appear in sessions. Currently the following extended values are supported in the JSON it dumps:

- Markup objects
- UUID objects
- datetime objects
- tuples

---

**Notice**

The PERMANENT_SESSION_LIFETIME config key can also be an integer starting with Flask 0.8. Either catch this down yourself or use the permanent_session_lifetime attribute on the app which converts the result to an integer automatically.

---

# Test Client

**class** flask.testing.**FlaskClient**(**args*, ***kwargs*)
Works like a regular Werkzeug test client but has some knowledge about how Flask works to defer the cleanup of the request context stack to the end of a with body when used in a with statement. For general information about how to use this class refer to werkzeug.test.Client.

Changed in version 0.12: *app.test_client()* includes preset default environment, which can be set after instantiation of the *app.test_client()* object in *client.environ_base*.

Basic usage is outlined in the *Testing Flask Applications* chapter.

**session_transaction**(**args*, ***kwargs*)

When used in combination with a `with` statement this opens a session transaction. This can be used to modify the session that the test client uses. Once the `with` block is left the session is stored back.

```
with client.session_transaction() as session:
    session['value'] = 42
```

Internally this is implemented by going through a temporary test request context and since session handling could depend on request variables this function accepts the same arguments as `test_request_context()` which are directly passed through.

## Application Globals

To share data that is valid for one request only from one function to another, a global variable is not good enough because it would break in threaded environments. Flask provides you with a special object that ensures it is only valid for the active request and that will return different values for each request. In a nutshell: it does the right thing, like it does for `request` and `session`.

`flask.`**g**

Just store on this whatever you want. For example a database connection or the user that is currently logged in.

Starting with Flask 0.10 this is stored on the application context and no longer on the request context which means it becomes available if only the application context is bound and not yet a request. This is especially useful when combined with the *Faking Resources and Context* pattern for testing.

Additionally as of 0.10 you can use the `get()` method to get an attribute or `None` (or the second argument) if it's not set. These two usages are now equivalent:

```
user = getattr(flask.g, 'user', None)
user = flask.g.get('user', None)
```

It's now also possible to use the `in` operator on it to see if an attribute is defined and it yields all keys on iteration.

As of 0.11 you can use `pop()` and `setdefault()` in the same way you would use them on a dictionary.

This is a proxy. See *Notes On Proxies* for more information.

## Useful Functions and Classes

flask.**current_app**
: Points to the application handling the request. This is useful for extensions that want to support multiple applications running side by side. This is powered by the application context and not by the request context, so you can change the value of this proxy by using the `app_context()` method.

  This is a proxy. See *Notes On Proxies* for more information.

flask.**has_request_context**()
: If you have code that wants to test if a request context is there or not this function can be used. For instance, you may want to take advantage of request information if the request object is available, but fail silently if it is unavailable.

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and has_request_context():
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

  Alternatively you can also just test any of the context bound objects (such as `request` or `g` for truthness):

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and request:
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

  New in version 0.7.

flask.**copy_current_request_context**(*f*)
: A helper function that decorates a function to retain the current request context. This is useful when working with greenlets. The moment the function is decorated a copy of the request context is created and then pushed when the function is called.

  Example:

```
import gevent
from flask import copy_current_request_context

@app.route('/')
def index():
    @copy_current_request_context
    def do_some_work():
```

```
        # do some work here, it can access flask.request like you
        # would otherwise in the view function.
        ...
    gevent.spawn(do_some_work)
    return 'Regular response'
```

New in version 0.10.

flask.**has_app_context**()

Works like has_request_context() but for the application context. You can also just do a boolean check on the current_app object instead.

New in version 0.9.

flask.**url_for**(*endpoint*, ***values*)

Generates a URL to the given endpoint with the method provided.

Variable arguments that are unknown to the target endpoint are appended to the generated URL as query arguments. If the value of a query argument is None, the whole pair is skipped. In case blueprints are active you can shortcut references to the same blueprint by prefixing the local endpoint with a dot (.).

This will reference the index function local to the current blueprint:

```
url_for('.index')
```

For more information, head over to the *Quickstart*.

To integrate applications, Flask has a hook to intercept URL build errors through Flask.url_build_error_handlers. The *url_for* function results in a BuildError when the current app does not have a URL for the given endpoint and values. When it does, the current_app calls its url_build_error_handlers if it is not None, which can return a string to use as the result of *url_for* (instead of *url_for*'s default to raise the BuildError exception) or re-raise the exception. An example:

```
def external_url_handler(error, endpoint, values):
    "Looks up an external URL when `url_for` cannot build a URL."
    # This is an example of hooking the build_error_handler.
    # Here, lookup_url is some utility function you've built
    # which looks up the endpoint in some external URL registry.
    url = lookup_url(endpoint, **values)
    if url is None:
        # External lookup did not have a URL.
        # Re-raise the BuildError, in context of original traceback.
        exc_type, exc_value, tb = sys.exc_info()
        if exc_value is error:
            raise exc_type, exc_value, tb
        else:
            raise error
    # url_for will use this result, instead of raising BuildError.
    return url
```

```
app.url_build_error_handlers.append(external_url_handler)
```

Here, *error* is the instance of `BuildError`, and *endpoint* and *values* are the arguments passed into *url_for*. Note that this is for building URLs outside the current application, and not for handling 404 NotFound errors.

New in version 0.10: The *_scheme* parameter was added.

New in version 0.9: The *_anchor* and *_method* parameters were added.

New in version 0.9: Calls `Flask.handle_build_error()` on `BuildError`.

**Parameters**

- **`endpoint`** – the endpoint of the URL (name of the function)
- **`values`** – the variable arguments of the URL rule
- **`_external`** – if set to `True`, an absolute URL is generated. Server address can be changed via `SERVER_NAME` configuration variable which defaults to *localhost*.
- **`_scheme`** – a string specifying the desired URL scheme. The *_external* parameter must be set to `True` or a `ValueError` is raised. The default behavior uses the same scheme as the current request, or `PREFERRED_URL_SCHEME` from the *app configuration* if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.
- **`_anchor`** – if provided this is added as anchor to the URL.
- **`_method`** – if provided this explicitly specifies an HTTP method.

`flask.`**`abort`**(*status*, **args*, ***kwargs*)

Raises an `HTTPException` for the given status code or WSGI application:

```
abort(404)  # 404 Not Found
abort(Response('Hello World'))
```

Can be passed a WSGI application or a status code. If a status code is given it's looked up in the list of exceptions and will raise that exception, if passed a WSGI application it will wrap it in a proxy WSGI exception and raise that:

```
abort(404)
abort(Response('Hello World'))
```

`flask.`**`redirect`**(*location*, *code=302*, *Response=None*)

Returns a response object (a WSGI application) that, if called, redirects the client to the target location. Supported codes are 301, 302, 303, 305, and 307. 300 is not supported because it's not a real redirect and 304 because it's the answer for a request with a request with defined If-Modified-Since headers.

New in version 0.6: The location can now be a unicode string that is encoded using the `iri_to_uri()` function.

New in version 0.10: The class used for the Response object can now be passed in.

**Parameters**

- **location** – the location the response should redirect to.
- **code** – the redirect status code. defaults to 302.
- **Response** (class) – a Response class to use when instantiating a response. The default is `werkzeug.wrappers.Response` if unspecified.

`flask.`**make_response**(**args*)

Sometimes it is necessary to set additional headers in a view. Because views do not have to return response objects but can return a value that is converted into a response object by Flask itself, it becomes tricky to add headers to it. This function can be called instead of using a return and you will get a response object which you can use to attach headers.

If view looked like this and you want to add a new header:

```
def index():
    return render_template('index.html', foo=42)
```

You can now do something like this:

```
def index():
    response = make_response(render_template('index.html', foo=42))
    response.headers['X-Parachutes'] = 'parachutes are cool'
    return response
```

This function accepts the very same arguments you can return from a view function. This for example creates a response with a 404 error code:

```
response = make_response(render_template('not_found.html'), 404)
```

The other use case of this function is to force the return value of a view function into a response which is helpful with view decorators:

```
response = make_response(view_function())
response.headers['X-Parachutes'] = 'parachutes are cool'
```

Internally this function does the following things:

- if no arguments are passed, it creates a new response argument
- if one argument is passed, `flask.Flask.make_response()` is invoked with it.
- if more than one argument is passed, the arguments are passed to the `flask.Flask.make_response()` function as tuple.

New in version 0.6.

flask.**after_this_request**(*f*)

Executes a function after this request. This is useful to modify response objects. The function is passed the response object and has to return the same or a new one.

Example:

```
@app.route('/')
def index():
    @after_this_request
    def add_header(response):
        response.headers['X-Foo'] = 'Parachute'
        return response
    return 'Hello World!'
```

This is more useful if a function other than the view function wants to modify a response. For instance think of a decorator that wants to add some headers without converting the return value into a response object.

New in version 0.9.

flask.**send_file**(*filename_or_fp*, *mimetype=None*, *as_attachment=False*, *attachment_filename=None*, *add_etags=True*, *cache_timeout=None*, *conditional=False*, *last_modified=None*)

Sends the contents of a file to the client. This will use the most efficient method available and configured. By default it will try to use the WSGI server's file_wrapper support. Alternatively you can set the application's `use_x_sendfile` attribute to `True` to directly emit an `X-Sendfile` header. This however requires support of the underlying webserver for `X-Sendfile`.

By default it will try to guess the mimetype for you, but you can also explicitly provide one. For extra security you probably want to send certain files as attachment (HTML for instance). The mimetype guessing requires a *filename* or an *attachment_filename* to be provided.

ETags will also be attached automatically if a *filename* is provided. You can turn this off by setting *add_etags=False*.

If *conditional=True* and *filename* is provided, this method will try to upgrade the response stream to support range requests. This will allow the request to be answered with partial content response.

Please never pass filenames to this function from user sources; you should use `send_from_directory()` instead.

New in version 0.2.

New in version 0.5: The *add_etags*, *cache_timeout* and *conditional* parameters were added. The default behavior is now to attach etags.

Changed in version 0.7: mimetype guessing and etag support for file objects was deprecated because it was unreliable. Pass a filename if you are able to, otherwise attach an etag yourself. This functionality will be removed in Flask 1.0

Changed in version 0.9: cache_timeout pulls its default from application config, when None.

Changed in version 0.12: The filename is no longer automatically inferred from file objects. If you want to use automatic mimetype and etag support, pass a filepath via *filename_or_fp* or *attachment_filename*.

Changed in version 0.12: The *attachment_filename* is preferred over *filename* for MIME-type detection.

**Parameters**

- **filename_or_fp** – the filename of the file to send in *latin-1*. This is relative to the `root_path` if a relative path is specified. Alternatively a file object might be provided in which case `X-Sendfile` might not work and fall back to the traditional method. Make sure that the file pointer is positioned at the start of data to send before calling `send_file()`.
- **mimetype** – the mimetype of the file if provided. If a file path is given, auto detection happens as fallback, otherwise an error will be raised.
- **as_attachment** – set to `True` if you want to send this file with a `Content-Disposition: attachment` header.
- **attachment_filename** – the filename for the attachment if it differs from the file's filename.
- **add_etags** – set to `False` to disable attaching of etags.
- **conditional** – set to `True` to enable conditional responses.
- **cache_timeout** – the timeout in seconds for the headers. When `None` (default), this value is set by `get_send_file_max_age()` of `current_app`.
- **last_modified** – set the `Last-Modified` header to this value, a `datetime` or timestamp. If a file was passed, this overrides its mtime.

`flask.`**send_from_directory**(*directory*, *filename*, ***options*)

Send a file from a given directory with `send_file()`. This is a secure way to quickly expose static files from an upload folder or something similar.

Example usage:

```
@app.route('/uploads/<path:filename>')
def download_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename, as_attachment=True)
```

---

**Sending files and Performance**

It is strongly recommended to activate either `X-Sendfile` support in your webserver or (if no authentication happens) to tell the webserver to serve files for the given path on its own without calling into the web application for improved performance.

---

New in version 0.5.

**Parameters**

- **directory** – the directory where all the files are stored.
- **filename** – the filename relative to that directory to download.
- **options** – optional keyword arguments that are directly forwarded to `send_file()`.

`flask.`**safe_join**(*directory*, **pathnames*)

Safely join *directory* and zero or more untrusted *pathnames* components.

Example usage:

```
@app.route('/wiki/<path:filename>')
def wiki_page(filename):
    filename = safe_join(app.config['WIKI_FOLDER'], filename)
    with open(filename, 'rb') as fd:
        content = fd.read()  # Read and process the file content...
```

**Parameters**

- **directory** – the trusted base directory.
- **pathnames** – the untrusted pathnames relative to that directory.

**Raises** `NotFound` if one or more passed paths fall out of its boundaries.

`flask.`**escape**(*s*) → markup

Convert the characters &, <, >, ‘, and ” in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

**class** `flask.`**Markup**

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the *__html__* interface a couple of frameworks and web applications use. `Markup` is a direct subclass of *unicode* and provides all the methods of *unicode* just that it escapes arguments passed and always returns *Markup*.

The *escape* function returns markup objects so that double escaping can’t happen.

The constructor of the `Markup` class can be used for three different things: When passed an unicode object it’s assumed to be safe, when passed an object with an HTML representation (has an *__html__* method) that representation is used, otherwise the object passed is converted into a unicode string and then assumed to be safe:

```
>>> Markup("Hello <em>World</em>!")
Markup(u'Hello <em>World</em>!')
>>> class Foo(object):
...  def __html__(self):
...   return '<a href="#">foo</a>'
...
>>> Markup(Foo())
Markup(u'<a href="#">foo</a>')
```

If you want object passed being always treated as unsafe you can use the `escape()` classmethod to create a `Markup` object:

```
>>> Markup.escape("Hello <em>World</em>!")
Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')
```

Operations on a markup string are markup aware which means that all arguments are passed through the `escape()` function:

```
>>> em = Markup("<em>%s</em>")
>>> em % "foo & bar"
Markup(u'<em>foo &amp; bar</em>')
>>> strong = Markup("<strong>%(text)s</strong>")
>>> strong % {'text': '<blink>hacker here</blink>'}
Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup(u'<em>Hello</em> &lt;foo&gt;')
```

**classmethod escape**(*s*)
: Escape the string. Works like `escape()` with the difference that for subclasses of `Markup` this function would return the correct subclass.

**striptags**()
: Unescape markup into an text_type string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
u'Main \xbb About'
```

**unescape**()
: Unescape markup again into an text_type string. This also resolves known HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

# Message Flashing

flask.**flash**(*message*, *category='message'*)

Flashes a message to the next request. In order to remove the flashed message from the session and to display it to the user, the template has to call `get_flashed_messages()`.

Changed in version 0.3: *category* parameter added.

**Parameters**

- **message** – the message to be flashed.
- **category** – the category for the message. The following values are recommended: `'message'` for any kind of message, `'error'` for errors, `'info'` for information messages and `'warning'` for warnings. However any kind of string can be used as category.

flask.**get_flashed_messages**(*with_categories=False*, *category_filter=[]*)

Pulls all flashed messages from the session and returns them. Further calls in the same request to the function will return the same messages. By default just the messages are returned, but when *with_categories* is set to `True`, the return value will be a list of tuples in the form `(category, message)` instead.

Filter the flashed messages to one or more categories by providing those categories in *category_filter*. This allows rendering categories in separate html blocks. The *with_categories* and *category_filter* arguments are distinct:

- *with_categories* controls whether categories are returned with message text (`True` gives a tuple, where `False` gives just the message text).
- *category_filter* filters the messages down to only those matching the provided categories.

See *Message Flashing* for examples.

Changed in version 0.3: *with_categories* parameter added.

Changed in version 0.9: *category_filter* parameter added.

**Parameters**

- **with_categories** – set to `True` to also receive categories.
- **category_filter** – whitelist of categories to limit return values

# JSON Support

Flask uses `simplejson` for the JSON implementation. Since simplejson is provided by both the standard library as well as extension, Flask will try simplejson first and then fall back to the stdlib json module. On top of that it will delegate access to the current application's JSON encoders and decoders for easier customization.

So for starters instead of doing:

```
try:
    import simplejson as json
except ImportError:
    import json
```

You can instead just do this:

```
from flask import json
```

For usage examples, read the `json` documentation in the standard library. The following extensions are by default applied to the stdlib's JSON module:

1. `datetime` objects are serialized as **RFC 822** strings.
2. Any object with an `__html__` method (like `Markup`) will have that method called and then the return value is serialized as string.

The `htmlsafe_dumps()` function of this json module is also available as filter called `|tojson` in Jinja2. Note that inside `script` tags no escaping must take place, so make sure to disable escaping with `|safe` if you intend to use it inside `script` tags unless you are using Flask 0.10 which implies that:

```
<script type=text/javascript>
    doSomethingWith({{ user.username|tojson|safe }});
</script>
```

---

**Auto-Sort JSON Keys**

The configuration variable `JSON_SORT_KEYS` (*Configuration Handling*) can be set to false to stop Flask from auto-sorting keys. By default sorting is enabled and outside of the app context sorting is turned on.

Notice that disabling key sorting can cause issues when using content based HTTP caches and Python's hash randomization feature.

---

`flask.json.`**`jsonify`**(**args*, ***kwargs*)

This function wraps `dumps()` to add a few enhancements that make life easier. It turns the JSON output into a `Response` object with the `application/json` mimetype. For convenience, it also converts multiple arguments into an array or multiple keyword arguments into a dict. This means that both `jsonify(1,2,3)` and `jsonify([1,2,3])` serialize to `[1,2,3]`.

For clarity, the JSON serialization behavior has the following differences from `dumps()`:

1.Single argument: Passed straight through to `dumps()`.

2.Multiple arguments: Converted to an array before being passed to `dumps()`.

3. Multiple keyword arguments: Converted to a dict before being passed to `dumps()`.

4. Both args and kwargs: Behavior undefined and will throw an exception.

Example usage:

```
from flask import jsonify

@app.route('/_get_current_user')
def get_current_user():
    return jsonify(username=g.user.username,
                   email=g.user.email,
                   id=g.user.id)
```

This will send a JSON response like this to the browser:

```
{
    "username": "admin",
    "email": "admin@localhost",
    "id": 42
}
```

Changed in version 0.11: Added support for serializing top-level arrays. This introduces a security risk in ancient browsers. See *JSON Security* for details.

This function's response will be pretty printed if it was not requested with `X-Requested-With: XMLHttpRequest` to simplify debugging unless the `JSONIFY_PRETTYPRINT_REGULAR` config parameter is set to false. Compressed (not pretty) formatting currently means no indents and no spaces after separators.

New in version 0.2.

`flask.json.`**`dumps`**(*obj*, ***kwargs*)

Serialize `obj` to a JSON formatted `str` by using the application's configured encoder (`json_encoder`) if there is an application on the stack.

This function can return `unicode` strings or ascii-only bytestrings by default which coerce into unicode strings automatically. That behavior by default is controlled by the `JSON_AS_ASCII` configuration variable and can be overridden by the simplejson `ensure_ascii` parameter.

`flask.json.`**`dump`**(*obj*, *fp*, ***kwargs*)

Like `dumps()` but writes into a file object.

`flask.json.`**`loads`**(*s*, ***kwargs*)

Unserialize a JSON object from a string `s` by using the application's configured decoder (`json_decoder`) if there is an application on the stack.

`flask.json.`**`load`**(*fp*, ***kwargs*)

Like `loads()` but reads from a file object.

**class** flask.json.**JSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

The default Flask JSON encoder. This one extends the default simplejson encoder by also supporting `datetime` objects, `UUID` as well as `Markup` objects which are serialized as RFC 822 datetime strings (same as the HTTP date format). In order to support more data types override the `default()` method.

**default**(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    return JSONEncoder.default(self, o)
```

**class** flask.json.**JSONDecoder**(*encoding=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

The default JSON decoder. This one does not change the behavior from the default simplejson decoder. Consult the `json` documentation for more information. This decoder is not only used for the load functions of this module but also `Request`.

# Template Rendering

flask.**render_template**(*template_name_or_list*, ***context*)

Renders a template from the template folder with the given context.

**Parameters**

- **template_name_or_list** – the name of the template to be rendered, or an iterable with template names the first one existing will be rendered
- **context** – the variables that should be available in the context of the template.

flask.**render_template_string**(*source*, ***context*)

Renders a template from the given template source string with the given context.

Template variables will be autoescaped.

**Parameters**

- **source** – the source code of the template to be rendered
- **context** – the variables that should be available in the context of the template.

flask.**get_template_attribute**(*template_name*, *attribute*)

Loads a macro (or variable) a template exports. This can be used to invoke a macro from within Python code. If you for example have a template named `_cider.html` with the following contents:

```
{% macro hello(name) %}Hello {{ name }}!{% endmacro %}
```

You can access this from Python code like this:

```
hello = get_template_attribute('_cider.html', 'hello')
return hello('World')
```

New in version 0.2.

**Parameters**

- **template_name** – the name of the template
- **attribute** – the name of the variable of macro to access

# Configuration

**class** flask.**Config**(*root_path*, *defaults=None*)

Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

Either you can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Or alternatively you can define the configuration options in the module that calls `from_object()` or provide an import path to a module that should be loaded. It is also possible to tell it to use the same module and with that provide the configuration values just before the call:

```
DEBUG = True
SECRET_KEY = 'development key'
app.config.from_object(__name__)
```

In both cases (loading from any Python file or loading from modules), only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

```
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS X use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use *set* instead.

**Parameters**

- **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application's `root_path`.
- **defaults** – an optional dictionary of default values

**from_envvar**(*variable_name*, *silent=False*)

Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

**Parameters**

- **variable_name** – name of the environment variable
- **silent** – set to `True` if you want silent failure for missing files.

**Returns** bool. `True` if able to load config, `False` otherwise.

**from_json**(*filename*, *silent=False*)

Updates the values in the config from a JSON file. This function behaves as if the JSON object was a dictionary and passed to the `from_mapping()` function.

**Parameters**

- **filename** – the filename of the JSON file. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

New in version 0.11.

**from_mapping**(**mapping*, ***kwargs*)

Updates the config like `update()` ignoring items with non-upper keys.

New in version 0.11.

**from_object**(*obj*)

Updates the values from the given object. An object can be of one of the following two types:

- a string: in this case the object with that name will be imported
- an actual object reference: that object is used directly

Objects are usually either modules or classes. `from_object()` loads only the uppercase attributes of the module/class. A `dict` object will not work with `from_object()` because the keys of a `dict` are not attributes of the `dict` class.

Example of module-based configuration:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with `from_pyfile()` and ideally from a location not within the package because the package might be installed system wide.

See *Development / Production* for an example of class-based configuration using `from_object()`.

**Parameters** **obj** – an import name or object

**from_pyfile**(*filename*, *silent=False*)

Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the `from_object()` function.

**Parameters**

- **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

New in version 0.7: *silent* parameter.

**get_namespace**(*namespace*, *lowercase=True*, *trim_namespace=True*)

Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix. Example usage:

```
app.config['IMAGE_STORE_TYPE'] = 'fs'
app.config['IMAGE_STORE_PATH'] = '/var/app/images'
app.config['IMAGE_STORE_BASE_URL'] = 'http://img.website.com'
image_store_config = app.config.get_namespace('IMAGE_STORE_')
```

The resulting dictionary *image_store_config* would look like:

```
{
    'type': 'fs',
    'path': '/var/app/images',
```

```
    'base_url': 'http://img.website.com'
}
```

This is often useful when configuration options map directly to keyword arguments in functions or class constructors.

**Parameters**

- **namespace** – a configuration namespace
- **lowercase** – a flag indicating if the keys of the resulting dictionary should be lowercase
- **trim_namespace** – a flag indicating if the keys of the resulting dictionary should not include the namespace

New in version 0.11.

## Extensions

flask.**ext**

This module acts as redirect import module to Flask extensions. It was added in 0.8 as the canonical way to import Flask extensions and makes it possible for us to have more flexibility in how we distribute extensions.

If you want to use an extension named "Flask-Foo" you would import it from `ext` as follows:

```
from flask.ext import foo
```

New in version 0.8.

## Stream Helpers

flask.**stream_with_context**(*generator_or_function*)

Request contexts disappear when the response is started on the server. This is done for efficiency reasons and to make it less likely to encounter memory leaks with badly written WSGI middlewares. The downside is that if you are using streamed responses, the generator cannot access request bound information any more.

This function however can help you keep the context around for longer:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    @stream_with_context
```

```
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(generate())
```

Alternatively it can also be used around a specific generator:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

New in version 0.9.

## Useful Internals

**class** `flask.ctx.`**RequestContext**(*app*, *environ*, *request=None*)

The request context contains all request relevant information. It is created at the beginning of the request and pushed to the *_request_ctx_stack* and removed at the end of it. It will create the URL adapter and request object for the WSGI environment provided.

Do not attempt to use this class directly, instead use `test_request_context()` and `request_context()` to create this object.

When the request context is popped, it will evaluate all the functions registered on the application for teardown execution (`teardown_request()`).

The request context is automatically popped at the end of the request for you. In debug mode the request context is kept around if exceptions happen so that interactive debuggers have a chance to introspect the data. With 0.4 this can also be forced for requests that did not fail and outside of `DEBUG` mode. By setting `'flask._preserve_context'` to `True` on the WSGI environment the context will not pop itself at the end of the request. This is used by the `test_client()` for example to implement the deferred cleanup functionality.

You might find this helpful for unittests where you need the information from the context local around for a little longer. Make sure to properly `pop()` the stack yourself in that situation, otherwise your unittests will leak memory.

**copy**()

Creates a copy of this request context with the same request object. This can be used to move a request context to a different greenlet. Because the actual

request object is the same this cannot be used to move a request context to a different thread unless access to the request object is locked.

New in version 0.10.

**match_request**()
Can be overridden by a subclass to hook into the matching of the request.

**pop**(*exc=<object object>*)
Pops the request context and unbinds it by doing that. This will also trigger the execution of functions registered by the `teardown_request()` decorator.

Changed in version 0.9: Added the *exc* argument.

**push**()
Binds the request context to the current context.

flask.**_request_ctx_stack**
The internal `LocalStack` that is used to implement all the context local objects used in Flask. This is a documented instance and can be used by extensions and application code but the use is discouraged in general.

The following attributes are always present on each layer of the stack:

***app*** the active Flask application.

***url_adapter*** the URL adapter that was used to match the request.

***request*** the current request object.

***session*** the active session object.

*g* an object with all the attributes of the `flask.g` object.

***flashes*** an internal cache for the flashed messages.

Example usage:

```
from flask import _request_ctx_stack

def get_session():
    ctx = _request_ctx_stack.top
    if ctx is not None:
        return ctx.session
```

**class** flask.ctx.**AppContext**(*app*)
The application context binds an application object implicitly to the current thread or greenlet, similar to how the `RequestContext` binds request information. The application context is also implicitly created if a request context is created but the application is not on top of the individual application context.

**pop**(*exc=<object object>*)
Pops the app context.

**push**()
Binds the app context to the current context.

flask.**_app_ctx_stack**
Works similar to the request context but only binds the application. This is mainly there for extensions to store data.

New in version 0.9.

**class** flask.blueprints.**BlueprintSetupState**(*blueprint*, *app*, *options*, *first_registration*)
Temporary holder object for registering a blueprint with the application. An instance of this class is created by the make_setup_state() method and later passed to all register callback functions.

**add_url_rule**(*rule*, *endpoint=None*, *view_func=None*, ***options*)
A helper method to register a rule (and optionally a view function) to the application. The endpoint is automatically prefixed with the blueprint's name.

**app = None**
a reference to the current application

**blueprint = None**
a reference to the blueprint that created this setup state.

**first_registration = None**
as blueprints can be registered multiple times with the application and not everything wants to be registered multiple times on it, this attribute can be used to figure out if the blueprint was registered in the past already.

**options = None**
a dictionary with all options that were passed to the register_blueprint() method.

**subdomain = None**
The subdomain that the blueprint should be active for, None otherwise.

**url_defaults = None**
A dictionary with URL defaults that is added to each and every URL that was defined with the blueprint.

**url_prefix = None**
The prefix that should be used for all URLs defined on the blueprint.

# Signals

New in version 0.6.

signals.**signals_available**
True if the signaling system is available. This is the case when blinker is installed.

The following signals exist in Flask:

flask.**template_rendered**
This signal is sent when a template was successfully rendered. The signal is in-

voked with the instance of the template as *template* and the context as dictionary (named *context*).

Example subscriber:

```
def log_template_renders(sender, template, context, **extra):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
                        context)

from flask import template_rendered
template_rendered.connect(log_template_renders, app)
```

flask.**before_render_template**

This signal is sent before template rendering process. The signal is invoked with the instance of the template as *template* and the context as dictionary (named *context*).

Example subscriber:

```
def log_template_renders(sender, template, context, **extra):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
                        context)

from flask import before_render_template
before_render_template.connect(log_template_renders, app)
```

flask.**request_started**

This signal is sent when the request context is set up, before any request processing happens. Because the request context is already bound, the subscriber can access the request with the standard global proxies such as `request`.

Example subscriber:

```
def log_request(sender, **extra):
    sender.logger.debug('Request context is set up')

from flask import request_started
request_started.connect(log_request, app)
```

flask.**request_finished**

This signal is sent right before the response is sent to the client. It is passed the response to be sent named *response*.

Example subscriber:

```
def log_response(sender, response, **extra):
    sender.logger.debug('Request context is about to close down.  '
                        'Response: %s', response)

from flask import request_finished
```

```
request_finished.connect(log_response, app)
```

flask.**got_request_exception**

This signal is sent when an exception happens during request processing. It is sent *before* the standard exception handling kicks in and even in debug mode, where no exception handling happens. The exception itself is passed to the subscriber as *exception*.

Example subscriber:

```
def log_exception(sender, exception, **extra):
    sender.logger.debug('Got exception during processing: %s', exception)

from flask import got_request_exception
got_request_exception.connect(log_exception, app)
```

flask.**request_tearing_down**

This signal is sent when the request is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

```
def close_db_connection(sender, **extra):
    session.close()

from flask import request_tearing_down
request_tearing_down.connect(close_db_connection, app)
```

As of Flask 0.9, this will also be passed an *exc* keyword argument that has a reference to the exception that caused the teardown if there was one.

flask.**appcontext_tearing_down**

This signal is sent when the app context is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

```
def close_db_connection(sender, **extra):
    session.close()

from flask import appcontext_tearing_down
appcontext_tearing_down.connect(close_db_connection, app)
```

This will also be passed an *exc* keyword argument that has a reference to the exception that caused the teardown if there was one.

flask.**appcontext_pushed**

This signal is sent when an application context is pushed. The sender is the

application. This is usually useful for unittests in order to temporarily hook in information. For instance it can be used to set a resource early onto the *g* object.

Example usage:

```
from contextlib import contextmanager
from flask import appcontext_pushed

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
    with appcontext_pushed.connected_to(handler, app):
        yield
```

And in the testcode:

```
def test_user_me(self):
    with user_set(app, 'john'):
        c = app.test_client()
        resp = c.get('/users/me')
        assert resp.data == 'username=john'
```

New in version 0.10.

flask.**appcontext_popped**
This signal is sent when an application context is popped. The sender is the application. This usually falls in line with the appcontext_tearing_down signal.

New in version 0.10.

flask.**message_flashed**
This signal is sent when the application is flashing a message. The messages is sent as *message* keyword argument and the category as *category*.

Example subscriber:

```
recorded = []
def record(sender, message, category, **extra):
    recorded.append((message, category))

from flask import message_flashed
message_flashed.connect(record, app)
```

New in version 0.10.

**class** signals.**Namespace**
An alias for blinker.base.Namespace if blinker is available, otherwise a dummy class that creates fake signals. This class is available for Flask extensions that want to provide the same fallback system as Flask itself.

**signal**(*name*, *doc=None*)
Creates a new signal for this namespace if blinker is available, otherwise

returns a fake signal that has a send method that will do nothing but will fail with a `RuntimeError` for all other operations, including connecting.

## Class-Based Views

New in version 0.7.

**class** `flask.views.`**`View`**

Alternative way to use view functions. A subclass has to implement `dispatch_request()` which is called with the view arguments from the URL routing system. If `methods` is provided the methods do not have to be passed to the `add_url_rule()` method explicitly:

```
class MyView(View):
    methods = ['GET']

    def dispatch_request(self, name):
        return 'Hello %s!' % name

app.add_url_rule('/hello/<name>', view_func=MyView.as_view('myview'))
```

When you want to decorate a pluggable view you will have to either do that when the view function is created (by wrapping the return value of `as_view()`) or you can use the `decorators` attribute:

```
class SecretView(View):
    methods = ['GET']
    decorators = [superuser_required]

    def dispatch_request(self):
        ...
```

The decorators stored in the decorators list are applied one after another when the view function is created. Note that you can *not* use the class based decorators since those would decorate the view class and not the generated view function!

**classmethod** **`as_view`**(*name*, **class_args*, ***class_kwargs*)

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

**`decorators`** **`= ()`**

The canonical way to decorate class-based views is to decorate the return value of as_view(). However since this moves parts of the logic from the class declaration to the place where it's hooked into the routing system.

You can place one or more decorators in this list and whenever the view function is created the result is automatically decorated.

New in version 0.8.

**dispatch_request**()
Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

**methods = None**
A list of methods this view can handle.

**class** flask.views.**MethodView**
Like a regular class-based view but that dispatches requests to particular methods. For instance if you implement a method called `get()` it means it will respond to `'GET'` requests and the `dispatch_request()` implementation will automatically forward your request to that. Also `options` is set for you automatically:

```
class CounterAPI(MethodView):

    def get(self):
        return session.get('counter', 0)

    def post(self):
        session['counter'] = session.get('counter', 0) + 1
        return 'OK'

app.add_url_rule('/counter', view_func=CounterAPI.as_view('counter'))
```

# URL Route Registrations

Generally there are three ways to define rules for the routing system:

1. You can use the `flask.Flask.route()` decorator.
2. You can use the `flask.Flask.add_url_rule()` function.
3. You can directly access the underlying Werkzeug routing system which is exposed as `flask.Flask.url_map`.

Variable parts in the route can be specified with angular brackets (`/user/<username>`). By default a variable part in the URL accepts any string without a slash however a different converter can be specified as well by using `<converter:name>`.

Variable parts are passed to the view function as keyword arguments.

The following converters are available:

| | |
|---|---|
| *string* | accepts any text without a slash (the default) |
| *int* | accepts integers |
| *float* | like *int* but for floating point values |
| *path* | like the default but also accepts slashes |
| *any* | matches one of the items provided |
| *uuid* | accepts UUID strings |

Custom converters can be defined using `flask.Flask.url_map`.

Here are some examples:

```
@app.route('/')
def index():
    pass

@app.route('/<username>')
def show_user(username):
    pass

@app.route('/post/<int:post_id>')
def show_post(post_id):
    pass
```

An important detail to keep in mind is how Flask deals with trailing slashes. The idea is to keep each URL unique so the following rules apply:

1. If a rule ends with a slash and is requested without a slash by the user, the user is automatically redirected to the same page with a trailing slash attached.
2. If a rule does not end with a trailing slash and the user requests the page with a trailing slash, a 404 not found is raised.

This is consistent with how web servers deal with static files. This also makes it possible to use relative link targets safely.

You can also define multiple rules for the same function. They have to be unique however. Defaults can also be specified. Here for example is a definition for a URL that accepts an optional page:

```
@app.route('/users/', defaults={'page': 1})
@app.route('/users/page/<int:page>')
def show_users(page):
    pass
```

This specifies that `/users/` will be the URL for page one and `/users/page/N` will be the URL for page *N*.

Here are the parameters that `route()` and `add_url_rule()` accept. The only difference is that with the route parameter the view function is defined with the decorator instead of the *view_func* parameter.

| | |
|---|---|
| *rule* | the URL rule as string |
| *endpoint* | the endpoint for the registered URL rule. Flask itself assumes that the name of the view function is the name of the endpoint if not explicitly stated. |
| *view_func* | the function to call when serving a request to the provided endpoint. If this is not provided one can specify the function later by storing it in the `view_functions` dictionary with the endpoint as key. |
| *defaults* | A dictionary with defaults for this rule. See the example above for how defaults work. |
| *subdomain* | specifies the rule for the subdomain in case subdomain matching is in use. If not specified the default subdomain is assumed. |
| ***options* | the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling. They have to be specified as keyword arguments. |

## View Function Options

For internal usage the view functions can have some attributes attached to customize behavior the view function would normally not have control over. The following attributes can be provided optionally to either override some defaults to `add_url_rule()` or general behavior:

- *__name__*: The name of a function is by default used as endpoint. If endpoint is provided explicitly this value is used. Additionally this will be prefixed with the name of the blueprint by default which cannot be customized from the function itself.
- *methods*: If methods are not provided when the URL rule is added, Flask will look on the view function object itself if a *methods* attribute exists. If it does, it will pull the information for the methods from there.
- *provide_automatic_options*: if this attribute is set Flask will either force enable or disable the automatic implementation of the HTTP `OPTIONS` response. This can be useful when working with decorators that want to customize the `OPTIONS` response on a per-view basis.
- *required_methods*: if this attribute is set, Flask will always add these methods when registering a URL rule even if the methods were explicitly overridden in the `route()` call.

Full example:

```
def index():
    if request.method == 'OPTIONS':
        # custom options handling here
        ...
```

```
    return 'Hello World!'
index.provide_automatic_options = False
index.methods = ['GET', 'OPTIONS']

app.add_url_rule('/', index)
```

New in version 0.8: The *provide_automatic_options* functionality was added.

## Command Line Interface

**class** flask.cli.**FlaskGroup**(*add_default_commands=True*, *create_app=None*, *add_version_option=True*, ***extra*)

Special subclass of the AppGroup group that supports loading more commands from the configured Flask app. Normally a developer does not have to interface with this class but there are some very advanced use cases for which it makes sense to create an instance of this.

For information as of why this is useful see *Custom Scripts*.

**Parameters**

- **add_default_commands** – if this is True then the default run and shell commands wil be added.
- **add_version_option** – adds the `--version` option.
- **create_app** – an optional callback that is passed the script info and returns the loaded app.

**class** flask.cli.**AppGroup**(*name=None*, *commands=None*, ***attrs*)

This works similar to a regular click Group but it changes the behavior of the command() decorator so that it automatically wraps the functions in with_appcontext().

Not to be confused with FlaskGroup.

**command**(**args*, ***kwargs*)

This works exactly like the method of the same name on a regular click.Group but it wraps callbacks in with_appcontext() unless it's disabled by passing with_appcontext=False.

**group**(**args*, ***kwargs*)

This works exactly like the method of the same name on a regular click.Group but it defaults the group class to AppGroup.

**class** flask.cli.**ScriptInfo**(*app_import_path=None*, *create_app=None*)

Help object to deal with Flask applications. This is usually not necessary to interface with as it's used internally in the dispatching to click. In future versions of Flask this object will most likely play a bigger role. Typically it's created automatically by the FlaskGroup but you can also manually create it and pass it onwards as click object.

**app_import_path = None**
Optionally the import path for the Flask application.

**create_app = None**
Optionally a function that is passed the script info to create the instance of the application.

**data = None**
A dictionary with arbitrary data that can be associated with this script info.

**load_app**()
Loads the Flask app (if not yet loaded) and returns it. Calling this multiple times will just result in the already loaded app to be returned.

flask.cli.**with_appcontext**(*f*)
Wraps a callback so that it's guaranteed to be executed with the script's application context. If callbacks are registered directly to the `app.cli` object then they are wrapped with this function by default unless it's disabled.

flask.cli.**pass_script_info**(*f*)
Marks a function so that an instance of `ScriptInfo` is passed as first argument to the click callback.

flask.cli.**run_command = <click.core.Command object>**
Runs a local development server for the Flask application.

This local server is recommended for development purposes only but it can also be used for simple intranet deployments. By default it will not support any sort of concurrency at all to simplify debugging. This can be changed with the –with-threads option which will enable basic multithreading.

The reloader and debugger are by default enabled if the debug flag of Flask is enabled and disabled otherwise.

flask.cli.**shell_command = <click.core.Command object>**
Runs an interactive Python shell in the context of a given Flask application. The application will populate the default namespace of this shell according to it's configuration.

This is useful for executing small snippets of management code without having to manually configuring the application.

Part III

# ADDITIONAL NOTES

Design notes, legal information and changelog are here for the interested.

CHAPTER 24

# Design Decisions in Flask

If you are curious why Flask does certain things the way it does and not differently, this section is for you. This should give you an idea about some of the design decisions that may appear arbitrary and surprising at first, especially in direct comparison with other frameworks.

## The Explicit Application Object

A Python web application based on WSGI has to have one central callable object that implements the actual application. In Flask this is an instance of the `Flask` class. Each Flask application has to create an instance of this class itself and pass it the name of the module, but why can't Flask do that itself?

Without such an explicit application object the following code:

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

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

Would look like this instead:

```
from hypothetical_flask import route

@route('/')
```

```
def index():
    return 'Hello World!'
```

There are three major reasons for this. The most important one is that implicit application objects require that there may only be one instance at the time. There are ways to fake multiple applications with a single application object, like maintaining a stack of applications, but this causes some problems I won't outline here in detail. Now the question is: when does a microframework need more than one application at the same time? A good example for this is unittesting. When you want to test something it can be very helpful to create a minimal application to test specific behavior. When the application object is deleted everything it allocated will be freed again.

Another thing that becomes possible when you have an explicit object lying around in your code is that you can subclass the base class (`Flask`) to alter specific behavior. This would not be possible without hacks if the object were created ahead of time for you based on a class that is not exposed to you.

But there is another very important reason why Flask depends on an explicit instantiation of that class: the package name. Whenever you create a Flask instance you usually pass it *__name__* as package name. Flask depends on that information to properly load resources relative to your module. With Python's outstanding support for reflection it can then access the package to figure out where the templates and static files are stored (see `open_resource()`). Now obviously there are frameworks around that do not need any configuration and will still be able to load templates relative to your application module. But they have to use the current working directory for that, which is a very unreliable way to determine where the application is. The current working directory is process-wide and if you are running multiple applications in one process (which could happen in a webserver without you knowing) the paths will be off. Worse: many webservers do not set the working directory to the directory of your application but to the document root which does not have to be the same folder.

The third reason is "explicit is better than implicit". That object is your WSGI application, you don't have to remember anything else. If you want to apply a WSGI middleware, just wrap it and you're done (though there are better ways to do that so that you do not lose the reference to the application object `wsgi_app()`).

Furthermore this design makes it possible to use a factory function to create the application which is very helpful for unittesting and similar things (*Application Factories*).

## The Routing System

Flask uses the Werkzeug routing system which was designed to automatically order routes by complexity. This means that you can declare routes in arbitrary order and they will still work as expected. This is a requirement if you want to properly implement decorator based routing since decorators could be fired in undefined order when the application is split into multiple modules.

Another design decision with the Werkzeug routing system is that routes in Werkzeug

try to ensure that URLs are unique. Werkzeug will go quite far with that in that it will automatically redirect to a canonical URL if a route is ambiguous.

## One Template Engine

Flask decides on one template engine: Jinja2. Why doesn't Flask have a pluggable template engine interface? You can obviously use a different template engine, but Flask will still configure Jinja2 for you. While that limitation that Jinja2 is *always* configured will probably go away, the decision to bundle one template engine and use that will not.

Template engines are like programming languages and each of those engines has a certain understanding about how things work. On the surface they all work the same: you tell the engine to evaluate a template with a set of variables and take the return value as string.

But that's about where similarities end. Jinja2 for example has an extensive filter system, a certain way to do template inheritance, support for reusable blocks (macros) that can be used from inside templates and also from Python code, uses Unicode for all operations, supports iterative template rendering, configurable syntax and more. On the other hand an engine like Genshi is based on XML stream evaluation, template inheritance by taking the availability of XPath into account and more. Mako on the other hand treats templates similar to Python modules.

When it comes to connecting a template engine with an application or framework there is more than just rendering templates. For instance, Flask uses Jinja2's extensive autoescaping support. Also it provides ways to access macros from Jinja2 templates.

A template abstraction layer that would not take the unique features of the template engines away is a science on its own and a too large undertaking for a microframework like Flask.

Furthermore extensions can then easily depend on one template language being present. You can easily use your own templating language, but an extension could still depend on Jinja itself.

## Micro with Dependencies

Why does Flask call itself a microframework and yet it depends on two libraries (namely Werkzeug and Jinja2). Why shouldn't it? If we look over to the Ruby side of web development there we have a protocol very similar to WSGI. Just that it's called Rack there, but besides that it looks very much like a WSGI rendition for Ruby. But nearly all applications in Ruby land do not work with Rack directly, but on top of a library with the same name. This Rack library has two equivalents in Python: WebOb (formerly Paste) and Werkzeug. Paste is still around but from my understanding it's sort of deprecated in favour of WebOb. The development of WebOb and Werkzeug

started side by side with similar ideas in mind: be a good implementation of WSGI for other applications to take advantage.

Flask is a framework that takes advantage of the work already done by Werkzeug to properly interface WSGI (which can be a complex task at times). Thanks to recent developments in the Python package infrastructure, packages with dependencies are no longer an issue and there are very few reasons against having libraries that depend on others.

## Thread Locals

Flask uses thread local objects (context local objects in fact, they support greenlet contexts as well) for request, session and an extra object you can put your own things on (g). Why is that and isn't that a bad idea?

Yes it is usually not such a bright idea to use thread locals. They cause troubles for servers that are not based on the concept of threads and make large applications harder to maintain. However Flask is just not designed for large applications or asynchronous servers. Flask wants to make it quick and easy to write a traditional web application.

Also see the *Becoming Big* section of the documentation for some inspiration for larger applications based on Flask.

## What Flask is, What Flask is Not

Flask will never have a database layer. It will not have a form library or anything else in that direction. Flask itself just bridges to Werkzeug to implement a proper WSGI application and to Jinja2 to handle templating. It also binds to a few common standard library packages such as logging. Everything else is up for extensions.

Why is this the case? Because people have different preferences and requirements and Flask could not meet those if it would force any of this into the core. The majority of web applications will need a template engine in some sort. However not every application needs a SQL database.

The idea of Flask is to build a good foundation for all applications. Everything else is up to you or extensions.

CHAPTER **25**

# HTML/XHTML FAQ

The Flask documentation and example applications are using HTML5. You may notice that in many situations, when end tags are optional they are not used, so that the HTML is cleaner and faster to load. Because there is much confusion about HTML and XHTML among developers, this document tries to answer some of the major questions.

## History of XHTML

For a while, it appeared that HTML was about to be replaced by XHTML. However, barely any websites on the Internet are actual XHTML (which is HTML processed using XML rules). There are a couple of major reasons why this is the case. One of them is Internet Explorer's lack of proper XHTML support. The XHTML spec states that XHTML must be served with the MIME type `application/xhtml+xml`, but Internet Explorer refuses to read files with that MIME type. While it is relatively easy to configure Web servers to serve XHTML properly, few people do. This is likely because properly using XHTML can be quite painful.

One of the most important causes of pain is XML's draconian (strict and ruthless) error handling. When an XML parsing error is encountered, the browser is supposed to show the user an ugly error message, instead of attempting to recover from the error and display what it can. Most of the (X)HTML generation on the web is based on non-XML template engines (such as Jinja, the one used in Flask) which do not protect you from accidentally creating invalid XHTML. There are XML based template engines, such as Kid and the popular Genshi, but they often come with a larger runtime overhead and are not as straightforward to use because they have to obey XML rules.

The majority of users, however, assumed they were properly using XHTML. They wrote an XHTML doctype at the top of the document and self-closed all the necessary tags (<br> becomes <br/> or <br></br> in XHTML). However, even if the document properly validates as XHTML, what really determines XHTML/HTML processing in browsers is the MIME type, which as said before is often not set properly. So the valid XHTML was being treated as invalid HTML.

XHTML also changed the way JavaScript is used. To properly work with XHTML, programmers have to use the namespaced DOM interface with the XHTML namespace to query for HTML elements.

## History of HTML5

Development of the HTML5 specification was started in 2004 under the name "Web Applications 1.0" by the Web Hypertext Application Technology Working Group, or WHATWG (which was formed by the major browser vendors Apple, Mozilla, and Opera) with the goal of writing a new and improved HTML specification, based on existing browser behavior instead of unrealistic and backwards-incompatible specifications.

For example, in HTML4 `<title/Hello/` theoretically parses exactly the same as `<title>Hello</title>`. However, since people were using XHTML-like tags along the lines of `<link />`, browser vendors implemented the XHTML syntax over the syntax defined by the specification.

In 2007, the specification was adopted as the basis of a new HTML specification under the umbrella of the W3C, known as HTML5. Currently, it appears that XHTML is losing traction, as the XHTML 2 working group has been disbanded and HTML5 is being implemented by all major browser vendors.

## HTML versus XHTML

The following table gives you a quick overview of features available in HTML 4.01, XHTML 1.1 and HTML5. (XHTML 1.0 is not included, as it was superseded by XHTML 1.1 and the barely-used XHTML5.)

| | HTML4.01 | XHTML1.1 | HTML5 |
|---|---|---|---|
| `<tag/value/ == <tag>value</tag>` | ✔[1] | ✘ | ✘ |
| `<br/>` supported | ✘ | ✔ | ✔[2] |
| `<script/>` supported | ✘ | ✔ | ✘ |
| should be served as *text/html* | ✔ | ✘[3] | ✔ |
| should be served as *application/xhtml+xml* | ✘ | ✔ | ✘ |
| strict error handling | ✘ | ✔ | ✘ |
| inline SVG | ✘ | ✔ | ✔ |
| inline MathML | ✘ | ✔ | ✔ |
| `<video>` tag | ✘ | ✘ | ✔ |
| `<audio>` tag | ✘ | ✘ | ✔ |
| New semantic tags like `<article>` | ✘ | ✘ | ✔ |

## What does “strict” mean?

HTML5 has strictly defined parsing rules, but it also specifies exactly how a browser should react to parsing errors - unlike XHTML, which simply states parsing should abort. Some people are confused by apparently invalid syntax that still generates the expected results (for example, missing end tags or unquoted attribute values).

Some of these work because of the lenient error handling most browsers use when they encounter a markup error, others are actually specified. The following constructs are optional in HTML5 by standard, but have to be supported by browsers:

- Wrapping the document in an `<html>` tag
- Wrapping header elements in `<head>` or the body elements in `<body>`
- Closing the `<p>`, `<li>`, `<dt>`, `<dd>`, `<tr>`, `<td>`, `<th>`, `<tbody>`, `<thead>`, or `<tfoot>` tags.
- Quoting attributes, so long as they contain no whitespace or special characters (like <, >, ', or ").
- Requiring boolean attributes to have a value.

This means the following page in HTML5 is perfectly valid:

```
<!doctype html>
<title>Hello HTML5</title>
```

[1] This is an obscure feature inherited from SGML. It is usually not supported by browsers, for reasons detailed above.

[2] This is for compatibility with server code that generates XHTML for tags such as `<br>`. It should not be used in new code.

[3] XHTML 1.0 is the last XHTML standard that allows to be served as *text/html* for backwards compatibility reasons.

```
<div class=header>
  <h1>Hello HTML5</h1>
  <p class=tagline>HTML5 is awesome
</div>
<ul class=nav>
  <li><a href=/index>Index</a>
  <li><a href=/downloads>Downloads</a>
  <li><a href=/about>About</a>
</ul>
<div class=body>
  <h2>HTML5 is probably the future</h2>
  <p>
    There might be some other things around but in terms of
    browser vendor support, HTML5 is hard to beat.
  <dl>
    <dt>Key 1
    <dd>Value 1
    <dt>Key 2
    <dd>Value 2
  </dl>
</div>
```

## New technologies in HTML5

HTML5 adds many new features that make Web applications easier to write and to use.

- The `<audio>` and `<video>` tags provide a way to embed audio and video without complicated add-ons like QuickTime or Flash.
- Semantic elements like `<article>`, `<header>`, `<nav>`, and `<time>` that make content easier to understand.
- The `<canvas>` tag, which supports a powerful drawing API, reducing the need for server-generated images to present data graphically.
- New form control types like `<input type="date">` that allow user agents to make entering and validating values easier.
- Advanced JavaScript APIs like Web Storage, Web Workers, Web Sockets, geolocation, and offline applications.

Many other features have been added, as well. A good guide to new features in HTML5 is Mark Pilgrim's soon-to-be-published book, Dive Into HTML5. Not all of them are supported in browsers yet, however, so use caution.

## What should be used?

Currently, the answer is HTML5. There are very few reasons to use XHTML considering the latest developments in Web browsers. To summarize the reasons given above:

- Internet Explorer (which, sadly, currently leads in market share) has poor support for XHTML.
- Many JavaScript libraries also do not support XHTML, due to the more complicated namespacing API it requires.
- HTML5 adds several new features, including semantic tags and the long-awaited `<audio>` and `<video>` tags.
- It has the support of most browser vendors behind it.
- It is much easier to write, and more compact.

For most applications, it is undoubtedly better to use HTML5 than XHTML.

CHAPTER **26**

# Security Considerations

Web applications usually face all kinds of security problems and it's very hard to get everything right. Flask tries to solve a few of these things for you, but there are a couple more you have to take care of yourself.

## Cross-Site Scripting (XSS)

Cross site scripting is the concept of injecting arbitrary HTML (and with it JavaScript) into the context of a website. To remedy this, developers have to properly escape text so that it cannot include arbitrary HTML tags. For more information on that have a look at the Wikipedia article on Cross-Site Scripting.

Flask configures Jinja2 to automatically escape all values unless explicitly told otherwise. This should rule out all XSS problems caused in templates, but there are still other places where you have to be careful:

- generating HTML without the help of Jinja2
- calling `Markup` on data submitted by users
- sending out HTML from uploaded files, never do that, use the `Content-Disposition: attachment` header to prevent that problem.
- sending out textfiles from uploaded files. Some browsers are using content-type guessing based on the first few bytes so users could trick a browser to execute HTML.

Another thing that is very important are unquoted attributes. While Jinja2 can protect you from XSS issues by escaping HTML, there is one thing it cannot protect you from: XSS by attribute injection. To counter this possible attack vector, be sure to always

quote your attributes with either double or single quotes when using Jinja expressions in them:

```
<a href="{{ href }}">the text</a>
```

Why is this necessary? Because if you would not be doing that, an attacker could easily inject custom JavaScript handlers. For example an attacker could inject this piece of HTML+JavaScript:

```
onmouseover=alert(document.cookie)
```

When the user would then move with the mouse over the link, the cookie would be presented to the user in an alert window. But instead of showing the cookie to the user, a good attacker might also execute any other JavaScript code. In combination with CSS injections the attacker might even make the element fill out the entire page so that the user would just have to have the mouse anywhere on the page to trigger the attack.

## Cross-Site Request Forgery (CSRF)

Another big problem is CSRF. This is a very complex topic and I won't outline it here in detail just mention what it is and how to theoretically prevent it.

If your authentication information is stored in cookies, you have implicit state management. The state of "being logged in" is controlled by a cookie, and that cookie is sent with each request to a page. Unfortunately that includes requests triggered by 3rd party sites. If you don't keep that in mind, some people might be able to trick your application's users with social engineering to do stupid things without them knowing.

Say you have a specific URL that, when you sent `POST` requests to will delete a user's profile (say `http://example.com/user/delete`). If an attacker now creates a page that sends a post request to that page with some JavaScript they just have to trick some users to load that page and their profiles will end up being deleted.

Imagine you were to run Facebook with millions of concurrent users and someone would send out links to images of little kittens. When users would go to that page, their profiles would get deleted while they are looking at images of fluffy cats.

How can you prevent that? Basically for each request that modifies content on the server you would have to either use a one-time token and store that in the cookie **and** also transmit it with the form data. After receiving the data on the server again, you would then have to compare the two tokens and ensure they are equal.

Why does Flask not do that for you? The ideal place for this to happen is the form validation framework, which does not exist in Flask.

## JSON Security

In Flask 0.10 and lower, `jsonify()` did not serialize top-level arrays to JSON. This was because of a security vulnerability in ECMAScript 4.

ECMAScript 5 closed this vulnerability, so only extremely old browsers are still vulnerable. All of these browsers have other more serious vulnerabilities, so this behavior was changed and `jsonify()` now supports serializing arrays.

CHAPTER 27

# Unicode in Flask

Flask, like Jinja2 and Werkzeug, is totally Unicode based when it comes to text. Not only these libraries, also the majority of web related Python libraries that deal with text. If you don't know Unicode so far, you should probably read The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets. This part of the documentation just tries to cover the very basics so that you have a pleasant experience with Unicode related things.

## Automatic Conversion

Flask has a few assumptions about your application (which you can change of course) that give you basic and painless Unicode support:

- the encoding for text on your website is UTF-8
- internally you will always use Unicode exclusively for text except for literal strings with only ASCII character points.
- encoding and decoding happens whenever you are talking over a protocol that requires bytes to be transmitted.

So what does this mean to you?

HTTP is based on bytes. Not only the protocol, also the system used to address documents on servers (so called URIs or URLs). However HTML which is usually transmitted on top of HTTP supports a large variety of character sets and which ones are used, are transmitted in an HTTP header. To not make this too complex Flask just assumes that if you are sending Unicode out you want it to be UTF-8 encoded. Flask will do the encoding and setting of the appropriate headers for you.

The same is true if you are talking to databases with the help of SQLAlchemy or a similar ORM system. Some databases have a protocol that already transmits Unicode and if they do not, SQLAlchemy or your other ORM should take care of that.

## The Golden Rule

So the rule of thumb: if you are not dealing with binary data, work with Unicode. What does working with Unicode in Python 2.x mean?

- as long as you are using ASCII charpoints only (basically numbers, some special characters of latin letters without umlauts or anything fancy) you can use regular string literals (`'Hello World'`).
- if you need anything else than ASCII in a string you have to mark this string as Unicode string by prefixing it with a lowercase *u*. (like `u'Hänsel und Gretel'`)
- if you are using non-Unicode characters in your Python files you have to tell Python which encoding your file uses. Again, I recommend UTF-8 for this purpose. To tell the interpreter your encoding you can put the `# -*- coding: utf-8 -*-` into the first or second line of your Python source file.
- Jinja is configured to decode the template files from UTF-8. So make sure to tell your editor to save the file as UTF-8 there as well.

## Encoding and Decoding Yourself

If you are talking with a filesystem or something that is not really based on Unicode you will have to ensure that you decode properly when working with Unicode interface. So for example if you want to load a file on the filesystem and embed it into a Jinja2 template you will have to decode it from the encoding of that file. Here the old problem that text files do not specify their encoding comes into play. So do yourself a favour and limit yourself to UTF-8 for text files as well.

Anyways. To load such a file with Unicode you can use the built-in `str.decode()` method:

```
def read_file(filename, charset='utf-8'):
    with open(filename, 'r') as f:
        return f.read().decode(charset)
```

To go from Unicode into a specific charset such as UTF-8 you can use the `unicode.encode()` method:

```
def write_file(filename, contents, charset='utf-8'):
    with open(filename, 'w') as f:
        f.write(contents.encode(charset))
```

# Configuring Editors

Most editors save as UTF-8 by default nowadays but in case your editor is not configured to do this you have to change it. Here some common ways to set your editor to store as UTF-8:

- Vim: put `set enc=utf-8` to your `.vimrc` file.
- Emacs: either use an encoding cookie or put this into your `.emacs` file:

```
(prefer-coding-system 'utf-8)
(setq default-buffer-file-coding-system 'utf-8)
```

- Notepad++:
  1. Go to *Settings -> Preferences ...*
  2. Select the "New Document/Default Directory" tab
  3. Select "UTF-8 without BOM" as encoding

  It is also recommended to use the Unix newline format, you can select it in the same panel but this is not a requirement.

CHAPTER **28**

# Flask Extension Development

Flask, being a microframework, often requires some repetitive steps to get a third party library working. Because very often these steps could be abstracted to support multiple projects the Flask Extension Registry was created.

If you want to create your own Flask extension for something that does not exist yet, this guide to extension development will help you get your extension running in no time and to feel like users would expect your extension to behave.

## Anatomy of an Extension

Extensions are all located in a package called `flask_something` where "something" is the name of the library you want to bridge. So for example if you plan to add support for a library named *simplexml* to Flask, you would name your extension's package `flask_simplexml`.

The name of the actual extension (the human readable name) however would be something like "Flask-SimpleXML". Make sure to include the name "Flask" somewhere in that name and that you check the capitalization. This is how users can then register dependencies to your extension in their `setup.py` files.

Flask sets up a redirect package called `flask.ext` where users should import the extensions from. If you for instance have a package called `flask_something` users would import it as `flask.ext.something`. This is done to transition from the old namespace packages. See *Extension Import Transition* for more details.

But what do extensions look like themselves? An extension has to ensure that it works with multiple Flask application instances at once. This is a requirement because many people will use patterns like the *Application Factories* pattern to create their application

as needed to aid unittests and to support multiple configurations. Because of that it is crucial that your application supports that kind of behavior.

Most importantly the extension must be shipped with a `setup.py` file and registered on PyPI. Also the development checkout link should work so that people can easily install the development version into their virtualenv without having to download the library by hand.

Flask extensions must be licensed under a BSD, MIT or more liberal license to be able to be enlisted in the Flask Extension Registry. Keep in mind that the Flask Extension Registry is a moderated place and libraries will be reviewed upfront if they behave as required.

## "Hello Flaskext!"

So let's get started with creating such a Flask extension. The extension we want to create here will provide very basic support for SQLite3.

First we create the following folder structure:

```
flask-sqlite3/
    flask_sqlite3.py
    LICENSE
    README
```

Here's the contents of the most important files:

### setup.py

The next file that is absolutely required is the `setup.py` file which is used to install your Flask extension. The following contents are something you can work with:

```
"""
Flask-SQLite3
-------------

This is the description for that library
"""
from setuptools import setup


setup(
    name='Flask-SQLite3',
    version='1.0',
    url='http://example.com/flask-sqlite3/',
    license='BSD',
    author='Your Name',
    author_email='your-email@example.com',
```

```
    description='Very short description',
    long_description=__doc__,
    py_modules=['flask_sqlite3'],
    # if you would be using a package instead use packages instead
    # of py_modules:
    # packages=['flask_sqlite3'],
    zip_safe=False,
    include_package_data=True,
    platforms='any',
    install_requires=[
        'Flask'
    ],
    classifiers=[
        'Environment :: Web Environment',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        'Topic :: Software Development :: Libraries :: Python Modules'
    ]
)
```

That's a lot of code but you can really just copy/paste that from existing extensions and adapt.

### flask_sqlite3.py

Now this is where your extension code goes. But how exactly should such an extension look like? What are the best practices? Continue reading for some insight.

## Initializing Extensions

Many extensions will need some kind of initialization step. For example, consider an application that's currently connecting to SQLite like the documentation suggests (*Using SQLite 3 with Flask*). So how does the extension know the name of the application object?

Quite simple: you pass it to it.

There are two recommended ways for an extension to initialize:

initialization functions:

> If your extension is called *helloworld* you might have a function called `init_helloworld(app[, extra_args])` that initializes the extension for that application. It could attach before / after handlers etc.

classes:

Classes work mostly like initialization functions but can later be used to further change the behavior. For an example look at how the OAuth extension works: there is an *OAuth* object that provides some helper functions like *OAuth.remote_app* to create a reference to a remote application that uses OAuth.

What to use depends on what you have in mind. For the SQLite 3 extension we will use the class-based approach because it will provide users with an object that handles opening and closing database connections.

What's important about classes is that they encourage to be shared around on module level. In that case, the object itself must not under any circumstances store any application specific state and must be shareable between different application.

## The Extension Code

Here's the contents of the *flask_sqlite3.py* for copy/paste:

```
import sqlite3
from flask import current_app

# Find the stack on which we want to store the database connection.
# Starting with Flask 0.9, the _app_ctx_stack is the correct one,
# before that we need to use the _request_ctx_stack.
try:
    from flask import _app_ctx_stack as stack
except ImportError:
    from flask import _request_ctx_stack as stack


class SQLite3(object):

    def __init__(self, app=None):
        self.app = app
        if app is not None:
            self.init_app(app)

    def init_app(self, app):
        app.config.setdefault('SQLITE3_DATABASE', ':memory:')
        # Use the newstyle teardown_appcontext if it's available,
        # otherwise fall back to the request context
        if hasattr(app, 'teardown_appcontext'):
            app.teardown_appcontext(self.teardown)
        else:
            app.teardown_request(self.teardown)

    def connect(self):
        return sqlite3.connect(current_app.config['SQLITE3_DATABASE'])
```

```
    def teardown(self, exception):
        ctx = stack.top
        if hasattr(ctx, 'sqlite3_db'):
            ctx.sqlite3_db.close()

    @property
    def connection(self):
        ctx = stack.top
        if ctx is not None:
            if not hasattr(ctx, 'sqlite3_db'):
                ctx.sqlite3_db = self.connect()
            return ctx.sqlite3_db
```

So here's what these lines of code do:

1. The `__init__` method takes an optional app object and, if supplied, will call `init_app`.
2. The `init_app` method exists so that the `SQLite3` object can be instantiated without requiring an app object. This method supports the factory pattern for creating applications. The `init_app` will set the configuration for the database, defaulting to an in memory database if no configuration is supplied. In addition, the `init_app` method attaches the `teardown` handler. It will try to use the new-style app context handler and if it does not exist, falls back to the request context one.
3. Next, we define a `connect` method that opens a database connection.
4. Finally, we add a `connection` property that on first access opens the database connection and stores it on the context. This is also the recommended way to handling resources: fetch resources lazily the first time they are used.

   Note here that we're attaching our database connection to the top application context via `_app_ctx_stack.top`. Extensions should use the top context for storing their own information with a sufficiently complex name. Note that we're falling back to the `_request_ctx_stack.top` if the application is using an older version of Flask that does not support it.

So why did we decide on a class-based approach here? Because using our extension looks something like this:

```
from flask import Flask
from flask_sqlite3 import SQLite3

app = Flask(__name__)
app.config.from_pyfile('the-config.cfg')
db = SQLite3(app)
```

You can then use the database from views like this:

```
@app.route('/')
def show_all():
```

```
    cur = db.connection.cursor()
    cur.execute(...)
```

Likewise if you are outside of a request but you are using Flask 0.9 or later with the app context support, you can use the database in the same way:

```
with app.app_context():
    cur = db.connection.cursor()
    cur.execute(...)
```

At the end of the `with` block the teardown handles will be executed automatically.

Additionally, the `init_app` method is used to support the factory pattern for creating apps:

```
db = Sqlite3()
# Then later on.
app = create_app('the-config.cfg')
db.init_app(app)
```

Keep in mind that supporting this factory pattern for creating apps is required for approved flask extensions (described below).

---

**Note on `init_app`**

As you noticed, `init_app` does not assign `app` to `self`. This is intentional! Class based Flask extensions must only store the application on the object when the application was passed to the constructor. This tells the extension: I am not interested in using multiple applications.

When the extension needs to find the current application and it does not have a reference to it, it must either use the `current_app` context local or change the API in a way that you can pass the application explicitly.

---

## Using _app_ctx_stack

In the example above, before every request, a `sqlite3_db` variable is assigned to `_app_ctx_stack.top`. In a view function, this variable is accessible using the `connection` property of `SQLite3`. During the teardown of a request, the `sqlite3_db` connection is closed. By using this pattern, the *same* connection to the sqlite3 database is accessible to anything that needs it for the duration of the request.

If the `_app_ctx_stack` does not exist because the user uses an old version of Flask, it is recommended to fall back to `_request_ctx_stack` which is bound to a request.

## Teardown Behavior

*This is only relevant if you want to support Flask 0.6 and older*

Due to the change in Flask 0.7 regarding functions that are run at the end of the request your extension will have to be extra careful there if it wants to continue to support older versions of Flask. The following pattern is a good way to support both:

```
def close_connection(response):
    ctx = _request_ctx_stack.top
    ctx.sqlite3_db.close()
    return response

if hasattr(app, 'teardown_request'):
    app.teardown_request(close_connection)
else:
    app.after_request(close_connection)
```

Strictly speaking the above code is wrong, because teardown functions are passed the exception and typically don't return anything. However because the return value is discarded this will just work assuming that the code in between does not touch the passed parameter.

## Learn from Others

This documentation only touches the bare minimum for extension development. If you want to learn more, it's a very good idea to check out existing extensions on the Flask Extension Registry. If you feel lost there is still the mailinglist and the IRC channel to get some ideas for nice looking APIs. Especially if you do something nobody before you did, it might be a very good idea to get some more input. This not only to get an idea about what people might want to have from an extension, but also to avoid having multiple developers working on pretty much the same side by side.

Remember: good API design is hard, so introduce your project on the mailinglist, and let other developers give you a helping hand with designing the API.

The best Flask extensions are extensions that share common idioms for the API. And this can only work if collaboration happens early.

## Approved Extensions

Flask also has the concept of approved extensions. Approved extensions are tested as part of Flask itself to ensure extensions do not break on new releases. These approved extensions are listed on the Flask Extension Registry and marked appropriately. If you want your own extension to be approved you have to follow these guidelines:

0. An approved Flask extension requires a maintainer. In the event an extension author would like to move beyond the project, the project should find a new maintainer including full source hosting transition and PyPI access. If no maintainer is available, give access to the Flask core team.
1. An approved Flask extension must provide exactly one package or module named `flask_extensionname`.
2. It must ship a testing suite that can either be invoked with `make test` or `python setup.py test`. For test suites invoked with `make test` the extension has to ensure that all dependencies for the test are installed automatically. If tests are invoked with `python setup.py test`, test dependencies can be specified in the `setup.py` file. The test suite also has to be part of the distribution.
3. APIs of approved extensions will be checked for the following characteristics:
    - an approved extension has to support multiple applications running in the same Python process.
    - it must be possible to use the factory pattern for creating applications.
4. The license must be BSD/MIT/WTFPL licensed.
5. The naming scheme for official extensions is *Flask-ExtensionName* or *ExtensionName-Flask*.
6. Approved extensions must define all their dependencies in the `setup.py` file unless a dependency cannot be met because it is not available on PyPI.
7. The extension must have documentation that uses one of the two Flask themes for Sphinx documentation.
8. The setup.py description (and thus the PyPI description) has to link to the documentation, website (if there is one) and there must be a link to automatically install the development version (`PackageName==dev`).
9. The `zip_safe` flag in the setup script must be set to `False`, even if the extension would be safe for zipping.
10. An extension currently has to support Python 2.6 as well as Python 2.7

## Extension Import Transition

In early versions of Flask we recommended using namespace packages for Flask extensions, of the form `flaskext.foo`. This turned out to be problematic in practice because it meant that multiple `flaskext` packages coexist. Consequently we have recommended to name extensions `flask_foo` over `flaskext.foo` for a long time.

Flask 0.8 introduced a redirect import system as a compatibility aid for app developers: Importing `flask.ext.foo` would try `flask_foo` and `flaskext.foo` in that order.

As of Flask 0.11, most Flask extensions have transitioned to the new naming schema. The `flask.ext.foo` compatibility alias is still in Flask 0.11 but is now deprecated – you

should use `flask_foo`.

CHAPTER 29

# Pocoo Styleguide

The Pocoo styleguide is the styleguide for all Pocoo Projects, including Flask. This styleguide is a requirement for Patches to Flask and a recommendation for Flask extensions.

In general the Pocoo Styleguide closely follows **PEP 8** with some small differences and extensions.

## General Layout

**Indentation:** 4 real spaces. No tabs, no exceptions.

**Maximum line length:** 79 characters with a soft limit for 84 if absolutely necessary. Try to avoid too nested code by cleverly placing *break*, *continue* and *return* statements.

**Continuing long statements:** To continue a statement you can use backslashes in which case you should align the next line with the last dot or equal sign, or indent four spaces:

```
this_is_a_very_long(function_call, 'with many parameters') \
    .that_returns_an_object_with_an_attribute

MyModel.query.filter(MyModel.scalar > 120) \
             .order_by(MyModel.name.desc()) \
             .limit(10)
```

If you break in a statement with parentheses or braces, align to the braces:

```
this_is_a_very_long(function_call, 'with many parameters',
                    23, 42, 'and even more')
```

For lists or tuples with many items, break immediately after the opening brace:

```
items = [
    'this is the first', 'set of items', 'with more items',
    'to come in this line', 'like this'
]
```

**Blank lines:** Top level functions and classes are separated by two lines, everything else by one. Do not use too many blank lines to separate logical segments in code. Example:

```
def hello(name):
    print 'Hello %s!' % name


def goodbye(name):
    print 'See you %s.' % name


class MyClass(object):
    """This is a simple docstring"""

    def __init__(self, name):
        self.name = name

    def get_annoying_name(self):
        return self.name.upper() + '!!!!111'
```

# Expressions and Statements

**General whitespace rules:**

- No whitespace for unary operators that are not words (e.g.: -, ~ etc.) as well on the inner side of parentheses.
- Whitespace is placed between binary operators.

Good:

```
exp = -1.05
value = (item_value / item_count) * offset / exp
value = my_list[index]
value = my_dict['key']
```

Bad:

```
exp = - 1.05
value = ( item_value / item_count ) * offset / exp
value = (item_value/item_count)*offset/exp
value=( item_value/item_count ) * offset/exp
value = my_list[ index ]
value = my_dict ['key']
```

**Yoda statements are a no-go:** Never compare constant with variable, always variable with constant:

Good:

```
if method == 'md5':
    pass
```

Bad:

```
if 'md5' == method:
    pass
```

**Comparisons:**

- against arbitrary types: `==` and `!=`
- against singletons with `is` and `is not` (eg: `foo is not None`)
- never compare something with `True` or `False` (for example never do `foo == False`, do `not foo` instead)

**Negated containment checks:** use `foo not in bar` instead of `not foo in bar`

**Instance checks:** `isinstance(a, C)` instead of `type(A) is C`, but try to avoid instance checks in general. Check for features.

# Naming Conventions

- Class names: `CamelCase`, with acronyms kept uppercase (`HTTPWriter` and not `HttpWriter`)
- Variable names: `lowercase_with_underscores`
- Method and function names: `lowercase_with_underscores`
- Constants: `UPPERCASE_WITH_UNDERSCORES`
- precompiled regular expressions: `name_re`

Protected members are prefixed with a single underscore. Double underscores are reserved for mixin classes.

On classes with keywords, trailing underscores are appended. Clashes with builtins are allowed and **must not** be resolved by appending an underline to the variable name.

If the function needs to access a shadowed builtin, rebind the builtin to a different name instead.

**Function and method arguments:**

- class methods: `cls` as first parameter
- instance methods: `self` as first parameter
- lambdas for properties might have the first parameter replaced with `x` like in `display_name = property(lambda x: x.real_name or x.username)`

## Docstrings

**Docstring conventions:** All docstrings are formatted with reStructuredText as understood by Sphinx. Depending on the number of lines in the docstring, they are laid out differently. If it's just one line, the closing triple quote is on the same line as the opening, otherwise the text is on the same line as the opening quote and the triple quote that closes the string on its own line:

```
def foo():
    """This is a simple docstring"""


def bar():
    """This is a longer docstring with so much information in there
    that it spans three lines.  In this case the closing triple quote
    is on its own line.
    """
```

**Module header:** The module header consists of an utf-8 encoding declaration (if non ASCII letters are used, but it is recommended all the time) and a standard docstring:

```
# -*- coding: utf-8 -*-
"""
    package.module
    ~~~~~~~~~~~~~~

    A brief description goes here.

    :copyright: (c) YEAR by AUTHOR.
    :license: LICENSE_NAME, see LICENSE_FILE for more details.
"""
```

Please keep in mind that proper copyrights and license files are a requirement for approved Flask extensions.

## Comments

Rules for comments are similar to docstrings. Both are formatted with reStructuredText. If a comment is used to document an attribute, put a colon after the opening pound sign (#):

```
class User(object):
    #: the name of the user as unicode string
    name = Column(String)
    #: the sha1 hash of the password + inline salt
    pw_hash = Column(String)
```

CHAPTER **30**

# Python 3 Support

Flask, its dependencies, and most Flask extensions support Python 3. You should start using Python 3 for your next project, but there are a few things to be aware of.

You need to use Python 3.3 or higher. 3.2 and older are *not* supported.

You should use the latest versions of all Flask-related packages. Flask 0.10 and Werkzeug 0.9 were the first versions to introduce Python 3 support.

Python 3 changed how unicode and bytes are handled, which complicates how low level code handles HTTP data. This mainly affects WSGI middleware interacting with the WSGI `environ` data. Werkzeug wraps that information in high-level helpers, so encoding issues should not affect you.

The majority of the upgrade work is in the lower-level libraries like Flask and Werkzeug, not the high-level application code. For example, all of the examples in the Flask repository work on both Python 2 and 3 and did not require a single line of code changed.

CHAPTER 31

# Upgrading to Newer Releases

Flask itself is changing like any software is changing over time. Most of the changes are the nice kind, the kind where you don't have to change anything in your code to profit from a new release.

However every once in a while there are changes that do require some changes in your code or there are changes that make it possible for you to improve your own code quality by taking advantage of new features in Flask.

This section of the documentation enumerates all the changes in Flask from release to release and how you can change your code to have a painless updating experience.

Use the **pip** command to upgrade your existing Flask installation by providing the `--upgrade` parameter:

```
$ pip install --upgrade Flask
```

## Version 0.12

### Changes to send_file

The `filename` is no longer automatically inferred from file-like objects. This means that the following code will no longer automatically have `X-Sendfile` support, etag generation or MIME-type guessing:

```
response = send_file(open('/path/to/file.txt'))
```

Any of the following is functionally equivalent:

```
fname = '/path/to/file.txt'

# Just pass the filepath directly
response = send_file(fname)

# Set the MIME-type and ETag explicitly
response = send_file(open(fname), mimetype='text/plain')
response.set_etag(...)

# Set `attachment_filename` for MIME-type guessing
# ETag still needs to be manually set
response = send_file(open(fname), attachment_filename=fname)
response.set_etag(...)
```

The reason for this is that some file-like objects have a invalid or even misleading `name` attribute. Silently swallowing errors in such cases was not a satisfying solution.

Additionally the default of falling back to `application/octet-stream` has been restricted. If Flask can't guess one or the user didn't provide one, the function fails if no filename information was provided.

# Version 0.11

0.11 is an odd release in the Flask release cycle because it was supposed to be the 1.0 release. However because there was such a long lead time up to the release we decided to push out a 0.11 release first with some changes removed to make the transition easier. If you have been tracking the master branch which was 1.0 you might see some unexpected changes.

In case you did track the master branch you will notice that **`flask --app`** is removed now. You need to use the environment variable to specify an application.

## Debugging

Flask 0.11 removed the `debug_log_format` attribute from Flask applications. Instead the new `LOGGER_HANDLER_POLICY` configuration can be used to disable the default log handlers and custom log handlers can be set up.

## Error handling

The behavior of error handlers was changed. The precedence of handlers used to be based on the decoration/call order of `errorhandler()` and `register_error_handler()`, respectively. Now the inheritance hierarchy takes precedence and handlers for more specific exception classes are executed instead of more general ones. See *Error handlers* for specifics.

Trying to register a handler on an instance now raises `ValueError`.

---

**Note:** There used to be a logic error allowing you to register handlers only for exception *instances*. This was unintended and plain wrong, and therefore was replaced with the intended behavior of registering handlers only using exception classes and HTTP error codes.

---

### Templating

The `render_template_string()` function has changed to autoescape template variables by default. This better matches the behavior of `render_template()`.

### Extension imports

Extension imports of the form `flask.ext.foo` are deprecated, you should use `flask_foo`.

The old form still works, but Flask will issue a `flask.exthook.ExtDeprecationWarning` for each extension you import the old way. We also provide a migration utility called flask-ext-migrate that is supposed to automatically rewrite your imports for this.

## Version 0.10

The biggest change going from 0.9 to 0.10 is that the cookie serialization format changed from pickle to a specialized JSON format. This change has been done in order to avoid the damage an attacker can do if the secret key is leaked. When you upgrade you will notice two major changes: all sessions that were issued before the upgrade are invalidated and you can only store a limited amount of types in the session. The new sessions are by design much more restricted to only allow JSON with a few small extensions for tuples and strings with HTML markup.

In order to not break people's sessions it is possible to continue using the old session system by using the Flask-OldSessions extension.

Flask also started storing the `flask.g` object on the application context instead of the request context. This change should be transparent for you but it means that you now can store things on the `g` object when there is no request context yet but an application context. The old `flask.Flask.request_globals_class` attribute was renamed to `flask.Flask.app_ctx_globals_class`.

## Version 0.9

The behavior of returning tuples from a function was simplified. If you return a tuple it no longer defines the arguments for the response object you're creating, it's now always a tuple in the form `(response, status, headers)` where at least one item has to be provided. If you depend on the old behavior, you can add it easily by subclassing Flask:

```
class TraditionalFlask(Flask):
    def make_response(self, rv):
        if isinstance(rv, tuple):
            return self.response_class(*rv)
        return Flask.make_response(self, rv)
```

If you maintain an extension that was using `_request_ctx_stack` before, please consider changing to `_app_ctx_stack` if it makes sense for your extension. For instance, the app context stack makes sense for extensions which connect to databases. Using the app context stack instead of the request context stack will make extensions more readily handle use cases outside of requests.

## Version 0.8

Flask introduced a new session interface system. We also noticed that there was a naming collision between `flask.session` the module that implements sessions and `flask.session` which is the global session object. With that introduction we moved the implementation details for the session system into a new module called `flask.sessions`. If you used the previously undocumented session support we urge you to upgrade.

If invalid JSON data was submitted Flask will now raise a `BadRequest` exception instead of letting the default `ValueError` bubble up. This has the advantage that you no longer have to handle that error to avoid an internal server error showing up for the user. If you were catching this down explicitly in the past as `ValueError` you will need to change this.

Due to a bug in the test client Flask 0.7 did not trigger teardown handlers when the test client was used in a with statement. This was since fixed but might require some changes in your test suites if you relied on this behavior.

## Version 0.7

In Flask 0.7 we cleaned up the code base internally a lot and did some backwards incompatible changes that make it easier to implement larger applications with Flask. Because we want to make upgrading as easy as possible we tried to counter the problems arising from these changes by providing a script that can ease the transition.

The script scans your whole application and generates an unified diff with changes it assumes are safe to apply. However as this is an automated tool it won't be able to find all use cases and it might miss some. We internally spread a lot of deprecation warnings all over the place to make it easy to find pieces of code that it was unable to upgrade.

We strongly recommend that you hand review the generated patchfile and only apply the chunks that look good.

If you are using git as version control system for your project we recommend applying the patch with `path -p1 < patchfile.diff` and then using the interactive commit feature to only apply the chunks that look good.

To apply the upgrade script do the following:

1. Download the script: flask-07-upgrade.py
2. Run it in the directory of your application:

```
python flask-07-upgrade.py > patchfile.diff
```

3. Review the generated patchfile.
4. Apply the patch:

```
patch -p1 < patchfile.diff
```

5. If you were using per-module template folders you need to move some templates around. Previously if you had a folder named `templates` next to a blueprint named `admin` the implicit template path automatically was `admin/index.html` for a template file called `templates/index.html`. This no longer is the case. Now you need to name the template `templates/admin/index.html`. The tool will not detect this so you will have to do that on your own.

Please note that deprecation warnings are disabled by default starting with Python 2.7. In order to see the deprecation warnings that might be emitted you have to enabled them with the `warnings` module.

If you are working with windows and you lack the `patch` command line utility you can get it as part of various Unix runtime environments for windows including cygwin, msysgit or ming32. Also source control systems like svn, hg or git have builtin support for applying unified diffs as generated by the tool. Check the manual of your version control system for more information.

## Bug in Request Locals

Due to a bug in earlier implementations the request local proxies now raise a `RuntimeError` instead of an `AttributeError` when they are unbound. If you caught these exceptions with `AttributeError` before, you should catch them with `RuntimeError` now.

Additionally the `send_file()` function is now issuing deprecation warnings if you depend on functionality that will be removed in Flask 0.11. Previously it was possible to use etags and mimetypes when file objects were passed. This was unreliable and caused issues for a few setups. If you get a deprecation warning, make sure to update your application to work with either filenames there or disable etag attaching and attach them yourself.

Old code:

```
return send_file(my_file_object)
return send_file(my_file_object)
```

New code:

```
return send_file(my_file_object, add_etags=False)
```

## Upgrading to new Teardown Handling

We streamlined the behavior of the callbacks for request handling. For things that modify the response the `after_request()` decorators continue to work as expected, but for things that absolutely must happen at the end of request we introduced the new `teardown_request()` decorator. Unfortunately that change also made after-request work differently under error conditions. It's not consistently skipped if exceptions happen whereas previously it might have been called twice to ensure it is executed at the end of the request.

If you have database connection code that looks like this:

```
@app.after_request
def after_request(response):
    g.db.close()
    return response
```

You are now encouraged to use this instead:

```
@app.teardown_request
def after_request(exception):
    if hasattr(g, 'db'):
        g.db.close()
```

On the upside this change greatly improves the internal code flow and makes it easier to customize the dispatching and error handling. This makes it now a lot easier to write unit tests as you can prevent closing down of database connections for a while. You can take advantage of the fact that the teardown callbacks are called when the response context is removed from the stack so a test can query the database after request handling:

```
with app.test_client() as client:
    resp = client.get('/')
    # g.db is still bound if there is such a thing
```

```
# and here it's gone
```

## Manual Error Handler Attaching

While it is still possible to attach error handlers to `Flask.error_handlers` it's discouraged to do so and in fact deprecated. In general we no longer recommend custom error handler attaching via assignments to the underlying dictionary due to the more complex internal handling to support arbitrary exception classes and blueprints. See `Flask.errorhandler()` for more information.

The proper upgrade is to change this:

```
app.error_handlers[403] = handle_error
```

Into this:

```
app.register_error_handler(403, handle_error)
```

Alternatively you should just attach the function with a decorator:

```
@app.errorhandler(403)
def handle_error(e):
    ...
```

(Note that `register_error_handler()` is new in Flask 0.7)

## Blueprint Support

Blueprints replace the previous concept of "Modules" in Flask. They provide better semantics for various features and work better with large applications. The update script provided should be able to upgrade your applications automatically, but there might be some cases where it fails to upgrade. What changed?

- Blueprints need explicit names. Modules had an automatic name guessing scheme where the shortname for the module was taken from the last part of the import module. The upgrade script tries to guess that name but it might fail as this information could change at runtime.
- Blueprints have an inverse behavior for `url_for()`. Previously `.foo` told `url_for()` that it should look for the endpoint `foo` on the application. Now it means "relative to current module". The script will inverse all calls to `url_for()` automatically for you. It will do this in a very eager way so you might end up with some unnecessary leading dots in your code if you're not using modules.
- Blueprints do not automatically provide static folders. They will also no longer automatically export templates from a folder called `templates` next to their location however but it can be enabled from the constructor. Same with static files: if

you want to continue serving static files you need to tell the constructor explicitly the path to the static folder (which can be relative to the blueprint's module path).

- Rendering templates was simplified. Now the blueprints can provide template folders which are added to a general template searchpath. This means that you need to add another subfolder with the blueprint's name into that folder if you want `blueprintname/template.html` as the template name.

If you continue to use the `Module` object which is deprecated, Flask will restore the previous behavior as good as possible. However we strongly recommend upgrading to the new blueprints as they provide a lot of useful improvement such as the ability to attach a blueprint multiple times, blueprint specific error handlers and a lot more.

## Version 0.6

Flask 0.6 comes with a backwards incompatible change which affects the order of after-request handlers. Previously they were called in the order of the registration, now they are called in reverse order. This change was made so that Flask behaves more like people expected it to work and how other systems handle request pre- and post-processing. If you depend on the order of execution of post-request functions, be sure to change the order.

Another change that breaks backwards compatibility is that context processors will no longer override values passed directly to the template rendering function. If for example `request` is as variable passed directly to the template, the default context processor will not override it with the current request object. This makes it easier to extend context processors later to inject additional variables without breaking existing template not expecting them.

## Version 0.5

Flask 0.5 is the first release that comes as a Python package instead of a single module. There were a couple of internal refactoring so if you depend on undocumented internal details you probably have to adapt the imports.

The following changes may be relevant to your application:

- autoescaping no longer happens for all templates. Instead it is configured to only happen on files ending with `.html`, `.htm`, `.xml` and `.xhtml`. If you have templates with different extensions you should override the `select_jinja_autoescape()` method.
- Flask no longer supports zipped applications in this release. This functionality might come back in future releases if there is demand for this feature. Removing support for this makes the Flask internal code easier to understand and fixes a couple of small issues that make debugging harder than necessary.

- The `create_jinja_loader` function is gone. If you want to customize the Jinja loader now, use the `create_jinja_environment()` method instead.

## Version 0.4

For application developers there are no changes that require changes in your code. In case you are developing on a Flask extension however, and that extension has a unittest-mode you might want to link the activation of that mode to the new `TESTING` flag.

## Version 0.3

Flask 0.3 introduces configuration support and logging as well as categories for flashing messages. All these are features that are 100% backwards compatible but you might want to take advantage of them.

### Configuration Support

The configuration support makes it easier to write any kind of application that requires some sort of configuration. (Which most likely is the case for any application out there).

If you previously had code like this:

```
app.debug = DEBUG
app.secret_key = SECRET_KEY
```

You no longer have to do that, instead you can just load a configuration into the config object. How this works is outlined in *Configuration Handling*.

### Logging Integration

Flask now configures a logger for you with some basic and useful defaults. If you run your application in production and want to profit from automatic error logging, you might be interested in attaching a proper log handler. Also you can start logging warnings and errors into the logger when appropriately. For more information on that, read *Application Errors*.

### Categories for Flash Messages

Flash messages can now have categories attached. This makes it possible to render errors, warnings or regular messages differently for example. This is an opt-in feature because it requires some rethinking in the code.

Read all about that in the *Message Flashing* pattern.

CHAPTER **32**

# Flask Changelog

Here you can see the full list of changes between each Flask release.

## Version 0.13

Major release, unreleased

- Make *app.run()* into a noop if a Flask application is run from the development server on the command line. This avoids some behavior that was confusing to debug for newcomers.
- Change default configuration *JSONIFY_PRETTYPRINT_REGULAR=False*. jsonify() method returns compressed response by default, and pretty response in debug mode.

## Version 0.12.2

Released on May 16 2017

- Fix a bug in *safe_join* on Windows.

## Version 0.12.1

Bugfix release, released on March 31st 2017

- Prevent *flask run* from showing a NoAppException when an ImportError occurs within the imported application module.
- Fix encoding behavior of `app.config.from_pyfile` for Python 3. Fix `#2118`.
- Call *ctx.auto_pop* with the exception object instead of *None*, in the event that a *BaseException* such as *KeyboardInterrupt* is raised in a request handler.

## Version 0.12

Released on December 21st 2016, codename Punsch.

- the cli command now responds to *–version*.
- Mimetype guessing and ETag generation for file-like objects in `send_file` has been removed, as per issue `#104`. See pull request `#1849`.
- Mimetype guessing in `send_file` now fails loudly and doesn't fall back to `application/octet-stream`. See pull request `#1988`.
- Make `flask.safe_join` able to join multiple paths like `os.path.join` (pull request `#1730`).
- Revert a behavior change that made the dev server crash instead of returning a Internal Server Error (pull request `#2006`).
- Correctly invoke response handlers for both regular request dispatching as well as error handlers.
- Disable logger propagation by default for the app logger.
- Add support for range requests in `send_file`.
- `app.test_client` includes preset default environment, which can now be directly set, instead of per `client.get`.

## Version 0.11.2

Bugfix release, unreleased

- Fix crash when running under PyPy3, see pull request `#1814`.

## Version 0.11.1

Bugfix release, released on June 7th 2016.

- Fixed a bug that prevented `FLASK_APP=foobar/__init__.py` from working. See pull request `#1872`.

# Version 0.11

Released on May 29th 2016, codename Absinthe.

- Added support to serializing top-level arrays to `flask.jsonify()`. This introduces a security risk in ancient browsers. See *JSON Security* for details.
- Added before_render_template signal.
- Added ***kwargs* to `flask.Test.test_client()` to support passing additional keyword arguments to the constructor of `flask.Flask.test_client_class`.
- Added `SESSION_REFRESH_EACH_REQUEST` config key that controls the set-cookie behavior. If set to `True` a permanent session will be refreshed each request and get their lifetime extended, if set to `False` it will only be modified if the session actually modifies. Non permanent sessions are not affected by this and will always expire if the browser window closes.
- Made Flask support custom JSON mimetypes for incoming data.
- Added support for returning tuples in the form `(response, headers)` from a view function.
- Added `flask.Config.from_json()`.
- Added `flask.Flask.config_class`.
- Added `flask.Config.get_namespace()`.
- Templates are no longer automatically reloaded outside of debug mode. This can be configured with the new `TEMPLATES_AUTO_RELOAD` config key.
- Added a workaround for a limitation in Python 3.3's namespace loader.
- Added support for explicit root paths when using Python 3.3's namespace packages.
- Added **`flask`** and the `flask.cli` module to start the local debug server through the click CLI system. This is recommended over the old `flask.run()` method as it works faster and more reliable due to a different design and also replaces `Flask-Script`.
- Error handlers that match specific classes are now checked first, thereby allowing catching exceptions that are subclasses of HTTP exceptions (in `werkzeug.exceptions`). This makes it possible for an extension author to create exceptions that will by default result in the HTTP error of their choosing, but may be caught with a custom error handler if desired.
- Added `flask.Config.from_mapping()`.
- Flask will now log by default even if debug is disabled. The log format is now hardcoded but the default log handling can be disabled through the `LOGGER_HANDLER_POLICY` configuration key.
- Removed deprecated module functionality.

- Added the `EXPLAIN_TEMPLATE_LOADING` config flag which when enabled will instruct Flask to explain how it locates templates. This should help users debug when the wrong templates are loaded.
- Enforce blueprint handling in the order they were registered for template loading.
- Ported test suite to py.test.
- Deprecated `request.json` in favour of `request.get_json()`.
- Add "pretty" and "compressed" separators definitions in jsonify() method. Reduces JSON response size when JSONIFY_PRETTYPRINT_REGULAR=False by removing unnecessary white space included by default after separators.
- JSON responses are now terminated with a newline character, because it is a convention that UNIX text files end with a newline and some clients don't deal well when this newline is missing. See https://github.com/pallets/flask/pull/1262 – this came up originally as a part of https://github.com/kennethreitz/httpbin/issues/168
- The automatically provided `OPTIONS` method is now correctly disabled if the user registered an overriding rule with the lowercase-version `options` (issue `#1288`).
- `flask.json.jsonify` now supports the `datetime.date` type (pull request `#1326`).
- Don't leak exception info of already catched exceptions to context teardown handlers (pull request `#1393`).
- Allow custom Jinja environment subclasses (pull request `#1422`).
- `flask.g` now has `pop()` and `setdefault` methods.
- Turn on autoescape for `flask.templating.render_template_string` by default (pull request `#1515`).
- `flask.ext` is now deprecated (pull request `#1484`).
- `send_from_directory` now raises BadRequest if the filename is invalid on the server OS (pull request `#1763`).
- Added the `JSONIFY_MIMETYPE` configuration variable (pull request `#1728`).
- Exceptions during teardown handling will no longer leave bad application contexts lingering around.

## Version 0.10.2

(bugfix release, release date to be announced)

- Fixed broken *test_appcontext_signals()* test case.
- Raise an `AttributeError` in `flask.helpers.find_package()` with a useful message explaining why it is raised when a PEP 302 import hook is used without an *is_package()* method.

- Fixed an issue causing exceptions raised before entering a request or app context to be passed to teardown handlers.
- Fixed an issue with query parameters getting removed from requests in the test client when absolute URLs were requested.
- Made *@before_first_request* into a decorator as intended.
- Fixed an etags bug when sending a file streams with a name.
- Fixed *send_from_directory* not expanding to the application root path correctly.
- Changed logic of before first request handlers to flip the flag after invoking. This will allow some uses that are potentially dangerous but should probably be permitted.
- Fixed Python 3 bug when a handler from *app.url_build_error_handlers* reraises the *BuildError*.

## Version 0.10.1

(bugfix release, released on June 14th 2013)

- Fixed an issue where `|tojson` was not quoting single quotes which made the filter not work properly in HTML attributes. Now it's possible to use that filter in single quoted attributes. This should make using that filter with angular.js easier.
- Added support for byte strings back to the session system. This broke compatibility with the common case of people putting binary data for token verification into the session.
- Fixed an issue where registering the same method twice for the same endpoint would trigger an exception incorrectly.

## Version 0.10

Released on June 13th 2013, codename Limoncello.

- Changed default cookie serialization format from pickle to JSON to limit the impact an attacker can do if the secret key leaks. See *Version 0.10* for more information.
- Added `template_test` methods in addition to the already existing `template_filter` method family.
- Added `template_global` methods in addition to the already existing `template_filter` method family.
- Set the content-length header for x-sendfile.
- `tojson` filter now does not escape script blocks in HTML5 parsers.

- `tojson` used in templates is now safe by default due. This was allowed due to the different escaping behavior.
- Flask will now raise an error if you attempt to register a new function on an already used endpoint.
- Added wrapper module around simplejson and added default serialization of datetime objects. This allows much easier customization of how JSON is handled by Flask or any Flask extension.
- Removed deprecated internal `flask.session` module alias. Use `flask.sessions` instead to get the session module. This is not to be confused with `flask.session` the session proxy.
- Templates can now be rendered without request context. The behavior is slightly different as the `request`, `session` and `g` objects will not be available and blueprint's context processors are not called.
- The config object is now available to the template as a real global and not through a context processor which makes it available even in imported templates by default.
- Added an option to generate non-ascii encoded JSON which should result in less bytes being transmitted over the network. It's disabled by default to not cause confusion with existing libraries that might expect `flask.json.dumps` to return bytestrings by default.
- `flask.g` is now stored on the app context instead of the request context.
- `flask.g` now gained a `get()` method for not erroring out on non existing items.
- `flask.g` now can be used with the `in` operator to see what's defined and it now is iterable and will yield all attributes stored.
- `flask.Flask.request_globals_class` got renamed to `flask.Flask.app_ctx_globals_class` which is a better name to what it does since 0.10.
- *request*, *session* and *g* are now also added as proxies to the template context which makes them available in imported templates. One has to be very careful with those though because usage outside of macros might cause caching.
- Flask will no longer invoke the wrong error handlers if a proxy exception is passed through.
- Added a workaround for chrome's cookies in localhost not working as intended with domain names.
- Changed logic for picking defaults for cookie values from sessions to work better with Google Chrome.
- Added *message_flashed* signal that simplifies flashing testing.
- Added support for copying of request contexts for better working with greenlets.
- Removed custom JSON HTTP exception subclasses. If you were relying on them you can reintroduce them again yourself trivially. Using them however

is strongly discouraged as the interface was flawed.

- Python requirements changed: requiring Python 2.6 or 2.7 now to prepare for Python 3.3 port.
- Changed how the teardown system is informed about exceptions. This is now more reliable in case something handles an exception halfway through the error handling process.
- Request context preservation in debug mode now keeps the exception information around which means that teardown handlers are able to distinguish error from success cases.
- Added the `JSONIFY_PRETTYPRINT_REGULAR` configuration variable.
- Flask now orders JSON keys by default to not trash HTTP caches due to different hash seeds between different workers.
- Added *appcontext_pushed* and *appcontext_popped* signals.
- The builtin run method now takes the `SERVER_NAME` into account when picking the default port to run on.
- Added *flask.request.get_json()* as a replacement for the old *flask.request.json* property.

## Version 0.9

Released on July 1st 2012, codename Campari.

- The `flask.Request.on_json_loading_failed()` now returns a JSON formatted response by default.
- The `flask.url_for()` function now can generate anchors to the generated links.
- The `flask.url_for()` function now can also explicitly generate URL rules specific to a given HTTP method.
- Logger now only returns the debug log setting if it was not set explicitly.
- Unregister a circular dependency between the WSGI environment and the request object when shutting down the request. This means that environ `werkzeug.request` will be `None` after the response was returned to the WSGI server but has the advantage that the garbage collector is not needed on CPython to tear down the request unless the user created circular dependencies themselves.
- Session is now stored after callbacks so that if the session payload is stored in the session you can still modify it in an after request callback.
- The `flask.Flask` class will avoid importing the provided import name if it can (the required first parameter), to benefit tools which build Flask instances programmatically. The Flask class will fall back to using import on systems with custom module hooks, e.g. Google App Engine, or when the import name is inside a zip archive (usually a .egg) prior to Python 2.7.

- Blueprints now have a decorator to add custom template filters application wide, `flask.Blueprint.app_template_filter()`.
- The Flask and Blueprint classes now have a non-decorator method for adding custom template filters application wide, `flask.Flask.add_template_filter()` and `flask.Blueprint.add_app_template_filter()`.
- The `flask.get_flashed_messages()` function now allows rendering flashed message categories in separate blocks, through a `category_filter` argument.
- The `flask.Flask.run()` method now accepts `None` for *host* and *port* arguments, using default values when `None`. This allows for calling run using configuration values, e.g. `app.run(app.config.get('MYHOST'), app.config.get('MYPORT'))`, with proper behavior whether or not a config file is provided.
- The `flask.render_template()` method now accepts a either an iterable of template names or a single template name. Previously, it only accepted a single template name. On an iterable, the first template found is rendered.
- Added `flask.Flask.app_context()` which works very similar to the request context but only provides access to the current application. This also adds support for URL generation without an active request context.
- View functions can now return a tuple with the first instance being an instance of `flask.Response`. This allows for returning `jsonify(error="error msg"), 400` from a view function.
- `Flask` and `Blueprint` now provide a `get_send_file_max_age()` hook for subclasses to override behavior of serving static files from Flask when using `flask.Flask.send_static_file()` (used for the default static file handler) and `send_file()`. This hook is provided a filename, which for example allows changing cache controls by file extension. The default max-age for *send_file* and static files can be configured through a new `SEND_FILE_MAX_AGE_DEFAULT` configuration variable, which is used in the default *get_send_file_max_age* implementation.
- Fixed an assumption in sessions implementation which could break message flashing on sessions implementations which use external storage.
- Changed the behavior of tuple return values from functions. They are no longer arguments to the response object, they now have a defined meaning.
- Added `flask.Flask.request_globals_class` to allow a specific class to be used on creation of the `g` instance of each request.
- Added *required_methods* attribute to view functions to force-add methods on registration.
- Added `flask.after_this_request()`.
- Added `flask.stream_with_context()` and the ability to push contexts multiple times without producing unexpected behavior.

## Version 0.8.1

Bugfix release, released on July 1st 2012

- Fixed an issue with the undocumented *flask.session* module to not work properly on Python 2.5. It should not be used but did cause some problems for package managers.

## Version 0.8

Released on September 29th 2011, codename Rakija

- Refactored session support into a session interface so that the implementation of the sessions can be changed without having to override the Flask class.
- Empty session cookies are now deleted properly automatically.
- View functions can now opt out of getting the automatic OPTIONS implementation.
- HTTP exceptions and Bad Request errors can now be trapped so that they show up normally in the traceback.
- Flask in debug mode is now detecting some common problems and tries to warn you about them.
- Flask in debug mode will now complain with an assertion error if a view was attached after the first request was handled. This gives earlier feedback when users forget to import view code ahead of time.
- Added the ability to register callbacks that are only triggered once at the beginning of the first request. (`Flask.before_first_request()`)
- Malformed JSON data will now trigger a bad request HTTP exception instead of a value error which usually would result in a 500 internal server error if not handled. This is a backwards incompatible change.
- Applications now not only have a root path where the resources and modules are located but also an instance path which is the designated place to drop files that are modified at runtime (uploads etc.). Also this is conceptually only instance depending and outside version control so it's the perfect place to put configuration files etc. For more information see *Instance Folders*.
- Added the `APPLICATION_ROOT` configuration variable.
- Implemented `session_transaction()` to easily modify sessions from the test environment.
- Refactored test client internally. The `APPLICATION_ROOT` configuration variable as well as `SERVER_NAME` are now properly used by the test client as defaults.
- Added `flask.views.View.decorators` to support simpler decorating of pluggable (class-based) views.

- Fixed an issue where the test client if used with the "with" statement did not trigger the execution of the teardown handlers.
- Added finer control over the session cookie parameters.
- HEAD requests to a method view now automatically dispatch to the *get* method if no handler was implemented.
- Implemented the virtual `flask.ext` package to import extensions from.
- The context preservation on exceptions is now an integral component of Flask itself and no longer of the test client. This cleaned up some internal logic and lowers the odds of runaway request contexts in unittests.

## Version 0.7.3

Bugfix release, release date to be decided

- Fixed the Jinja2 environment's list_templates method not returning the correct names when blueprints or modules were involved.

## Version 0.7.2

Bugfix release, released on July 6th 2011

- Fixed an issue with URL processors not properly working on blueprints.

## Version 0.7.1

Bugfix release, released on June 29th 2011

- Added missing future import that broke 2.5 compatibility.
- Fixed an infinite redirect issue with blueprints.

## Version 0.7

Released on June 28th 2011, codename Grappa

- Added `make_default_options_response()` which can be used by subclasses to alter the default behavior for `OPTIONS` responses.
- Unbound locals now raise a proper `RuntimeError` instead of an `AttributeError`.
- Mimetype guessing and etag support based on file objects is now deprecated for `flask.send_file()` because it was unreliable. Pass filenames instead or attach your own etags and provide a proper mimetype by hand.

- Static file handling for modules now requires the name of the static folder to be supplied explicitly. The previous autodetection was not reliable and caused issues on Google's App Engine. Until 1.0 the old behavior will continue to work but issue dependency warnings.
- fixed a problem for Flask to run on jython.
- added a `PROPAGATE_EXCEPTIONS` configuration variable that can be used to flip the setting of exception propagation which previously was linked to `DEBUG` alone and is now linked to either `DEBUG` or `TESTING`.
- Flask no longer internally depends on rules being added through the *add_url_rule* function and can now also accept regular werkzeug rules added to the url map.
- Added an *endpoint* method to the flask application object which allows one to register a callback to an arbitrary endpoint with a decorator.
- Use Last-Modified for static file sending instead of Date which was incorrectly introduced in 0.6.
- Added *create_jinja_loader* to override the loader creation process.
- Implemented a silent flag for *config.from_pyfile*.
- Added *teardown_request* decorator, for functions that should run at the end of a request regardless of whether an exception occurred. Also the behavior for *after_request* was changed. It's now no longer executed when an exception is raised. See *Upgrading to new Teardown Handling*
- Implemented `flask.has_request_context()`
- Deprecated *init_jinja_globals*. Override the `create_jinja_environment()` method instead to achieve the same functionality.
- Added `flask.safe_join()`
- The automatic JSON request data unpacking now looks at the charset mimetype parameter.
- Don't modify the session on `flask.get_flashed_messages()` if there are no messages in the session.
- *before_request* handlers are now able to abort requests with errors.
- it is not possible to define user exception handlers. That way you can provide custom error messages from a central hub for certain errors that might occur during request processing (for instance database connection errors, timeouts from remote resources etc.).
- Blueprints can provide blueprint specific error handlers.
- Implemented generic *Pluggable Views* (class-based views).

## Version 0.6.1

Bugfix release, released on December 31st 2010

- Fixed an issue where the default `OPTIONS` response was not exposing all valid methods in the `Allow` header.
- Jinja2 template loading syntax now allows "./" in front of a template load path. Previously this caused issues with module setups.
- Fixed an issue where the subdomain setting for modules was ignored for the static folder.
- Fixed a security problem that allowed clients to download arbitrary files if the host server was a windows based operating system and the client uses backslashes to escape the directory the files where exposed from.

## Version 0.6

Released on July 27th 2010, codename Whisky

- after request functions are now called in reverse order of registration.
- OPTIONS is now automatically implemented by Flask unless the application explicitly adds 'OPTIONS' as method to the URL rule. In this case no automatic OPTIONS handling kicks in.
- static rules are now even in place if there is no static folder for the module. This was implemented to aid GAE which will remove the static folder if it's part of a mapping in the .yml file.
- the `config` is now available in the templates as *config*.
- context processors will no longer override values passed directly to the render function.
- added the ability to limit the incoming request data with the new `MAX_CONTENT_LENGTH` configuration value.
- the endpoint for the `flask.Module.add_url_rule()` method is now optional to be consistent with the function of the same name on the application object.
- added a `flask.make_response()` function that simplifies creating response object instances in views.
- added signalling support based on blinker. This feature is currently optional and supposed to be used by extensions and applications. If you want to use it, make sure to have blinker installed.
- refactored the way URL adapters are created. This process is now fully customizable with the `create_url_adapter()` method.

- modules can now register for a subdomain instead of just an URL prefix. This makes it possible to bind a whole module to a configurable subdomain.

## Version 0.5.2

Bugfix Release, released on July 15th 2010

- fixed another issue with loading templates from directories when modules were used.

## Version 0.5.1

Bugfix Release, released on July 6th 2010

- fixes an issue with template loading from directories when modules where used.

## Version 0.5

Released on July 6th 2010, codename Calvados

- fixed a bug with subdomains that was caused by the inability to specify the server name. The server name can now be set with the `SERVER_NAME` config key. This key is now also used to set the session cookie cross-subdomain wide.
- autoescaping is no longer active for all templates. Instead it is only active for `.html`, `.htm`, `.xml` and `.xhtml`. Inside templates this behavior can be changed with the `autoescape` tag.
- refactored Flask internally. It now consists of more than a single file.
- `flask.send_file()` now emits etags and has the ability to do conditional responses builtin.
- (temporarily) dropped support for zipped applications. This was a rarely used feature and led to some confusing behavior.
- added support for per-package template and static-file directories.
- removed support for *create_jinja_loader* which is no longer used in 0.5 due to the improved module support.
- added a helper function to expose files from any directory.

## Version 0.4

Released on June 18th 2010, codename Rakia

- added the ability to register application wide error handlers from modules.
- `after_request()` handlers are now also invoked if the request dies with an exception and an error handling page kicks in.
- test client has not the ability to preserve the request context for a little longer. This can also be used to trigger custom requests that do not pop the request stack for testing.
- because the Python standard library caches loggers, the name of the logger is configurable now to better support unittests.
- added `TESTING` switch that can activate unittesting helpers.
- the logger switches to `DEBUG` mode now if debug is enabled.

## Version 0.3.1

Bugfix release, released on May 28th 2010

- fixed a error reporting bug with `flask.Config.from_envvar()`
- removed some unused code from flask
- release does no longer include development leftover files (.git folder for themes, built documentation in zip and pdf file and some .pyc files)

## Version 0.3

Released on May 28th 2010, codename Schnaps

- added support for categories for flashed messages.
- the application now configures a `logging.Handler` and will log request handling exceptions to that logger when not in debug mode. This makes it possible to receive mails on server errors for example.
- added support for context binding that does not require the use of the with statement for playing in the console.
- the request context is now available within the with statement making it possible to further push the request context or pop it.
- added support for configurations.

## Version 0.2

Released on May 12th 2010, codename Jägermeister

- various bugfixes

- integrated JSON support
- added `get_template_attribute()` helper function.
- `add_url_rule()` can now also register a view function.
- refactored internal request dispatching.
- server listens on 127.0.0.1 by default now to fix issues with chrome.
- added external URL support.
- added support for `send_file()`
- module support and internal request handling refactoring to better support pluggable applications.
- sessions can be set to be permanent now on a per-session basis.
- better error reporting on missing secret keys.
- added support for Google Appengine.

## Version 0.1

First public preview release.

CHAPTER **33**

# License

Flask is licensed under a three clause BSD License. It basically means: do whatever you want with it as long as the copyright in Flask sticks around, the conditions are not modified and the disclaimer is present. Furthermore you must not use the names of the authors to promote derivatives of the software without written consent.

The full license text can be found below (*Flask License*). For the documentation and artwork different licenses apply.

## Authors

Flask is written and maintained by Armin Ronacher and various contributors:

### Development Lead

- Armin Ronacher <armin.ronacher@active-4.com>

### Patches and Suggestions

- Adam Zapletal
- Ali Afshar
- Chris Edgemon
- Chris Grindstaff

- Christopher Grebs
- Daniel Neuhäuser
- Dan Sully
- David Lord @davidism
- Edmond Burnett
- Florent Xicluna
- Georg Brandl
- Jeff Widman @jeffwidman
- Justin Quick
- Kenneth Reitz
- Keyan Pishdadian
- Marian Sigler
- Martijn Pieters
- Matt Campell
- Matthew Frazier
- Michael van Tellingen
- Ron DuPlain
- Sebastien Estienne
- Simon Sapin
- Stephane Wirtel
- Thomas Schranz
- Zhao Xiaohong

## General License Definitions

The following section contains the full license texts for Flask and the documentation.

- "AUTHORS" hereby refers to all the authors listed in the *Authors* section.
- The "*Flask License*" applies to all the source code shipped as part of Flask (Flask itself as well as the examples and the unittests) as well as documentation.
- The "*Flask Artwork License*" applies to the project's Horn-Logo.

# Flask License

Copyright (c) 2015 by Armin Ronacher and contributors. See AUTHORS for more details.

Some rights reserved.

Redistribution and use in source and binary forms of the software as well as documentation, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE AND DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Flask Artwork License

Copyright (c) 2010 by Armin Ronacher.

Some rights reserved.

This logo or a modified version may be used by anyone to refer to the Flask project, but does not indicate endorsement by the project.

Redistribution and use in source (the SVG file) and binary forms (rendered PNG files etc.) of the image, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice and this list of conditions.

- The names of the contributors to the Flask software (see AUTHORS) may not be used to endorse or promote products derived from this software without specific prior written permission.

Note: we would appreciate that you make the image a link to http://flask.pocoo.org/ if you use it on a web page.

# Index

## Symbols



## A



## B

## C



## D



## E



## F

## G



## H



## I



## J

## R



## S

## T



## U



## V



## W



## Y