Flask Documentation

Release 0.1

Armin Ronacher

Jul 12, 2017

Contents

1 Textual Documentation **3**
1.1 Foreword . . . 3
1.2 Installation . . . 4
1.3 Quickstart . . . 6
1.4 Tutorial . . . 16
1.5 Testing Flask Applications . . . 23
1.6 Patterns for Flask . . . 26
1.7 Deployment Options . . . 33
1.8 Becoming Big . . . 37

2 Reference **39**
2.1 API . . . 39

Python Module Index **51**

Welcome to Flask's documentation. This documentation is divided in different parts. I would suggest to get started with the *Installation* and then heading over to the *Quickstart*. Besides the quickstart there is also a more detailed *Tutorial* that shows how to create a complete (albeit small) application with Flask. If you rather want to dive into all the internal parts of Flask, check out the *API* documentation. Common patterns are described in the *Patterns for Flask* section.

Flask also depends on two external libraries: the Jinja2 template engine and the Werkzeug WSGI toolkit. both of which are not documented here. If you want to dive into their documentation check out the following links:

- Jinja2 Documentation
- Werkzeug Documentation

CHAPTER 1

Textual Documentation

This part of the documentation is written text and should give you an idea how to work with Flask. It's a series of step-by-step instructions for web development.

Foreword

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

What does Micro Mean?

The micro in microframework for me means on the one hand being small in size and complexity but on the other hand also that the complexity of the applications that are written with these frameworks do not exceed a certain size. A microframework like Flask sacrifices a few things in order to be approachable and to be as concise as possible.

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. While this is a really easy approach and saves you a lot of time, it also does not scale well to large applications. It's especially painful for more complex unittests and when you suddenly have to deal with code being executed outside of the context of a request (for example if you have cronjobs).

Flask provides some tools to deal with the downsides of this approach but the core problem of this approach obviously stays. It is also based on convention over configuration which means that a lot of things are preconfigured in Flask and will work well for smaller applications but not so much for larger ones (where and how it looks for templates, static files etc.)

But don't worry if your application suddenly grows larger than it was initially and you're afraid Flask might not grow with it. Even with larger frameworks you sooner or later will find out that you need something the framework just cannot do for you without modification. If you are ever in that situation, check out the *Becoming Big* chapter.

A Framework and An Example

Flask is not only a microframework, it is also an example. Based on Flask, there will be a series of blog posts that explain how to create a framework. Flask itself is just one way to implement a framework on top of existing libraries. Unlike many other microframeworks Flask does not try to implement anything on its own, it reuses existing code.

Web Development is Dangerous

I'm not even joking. Well, maybe a little. 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 in a secure manner.

Unfortunately there are many ways 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.

Whenever something is dangerous where you have to watch out, the documentation will tell you so. Some of the security concerns of web development are far more complex than one might think and often we all end up in situations where we think "well, this is just far fetched, how could that possibly be exploited" and then an intelligent guy comes along and figures a way out to exploit that application. And don't think, your application is not important enough for hackers to take notice. Depending ont he kind of attack, chances are there are automated botnets out there trying to figure out how to fill your database with viagra adverisments.

So always keep that in mind when doing web development.

Target Audience

Is Flask for you? Is your application small-ish (less than 4000 lines of Python code) and does not depend on too complex database structures, Flask is the Framework for you. It was designed from the ground up to be easy to use, based on established principles, good intentions and on top of two established libraries in widespread usage.

Flask serves two purposes: it's an example of how to create a minimal and opinionated framework on top of Werkzeug to show how this can be done, and to provide people with a simple tool to prototype larger applications or to implement small and medium sized applications.

If you suddenly discover that your application grows larger than originally intended, head over to the *Becoming Big* section to see some possible solutions for larger applications.

Satisfied? Then head over to the *Installation*.

Installation

Flask is a microframework and yet it depends on external libraries. There are various ways how you can install that library and this explains each way and why there are multiple ways.

Flask depends on two external libraries: Werkzeug and Jinja2. The first on is responsible for interfacing WSGI the latter to render templates. Now you are maybe asking, what is WSGI? WSGI is a standard in Python that is basically responsible for ensuring that your application is behaving in a specific way that you can run it on different environments (for example on a local development server, on an Apache2, on lighttpd, on Google's App Engine or whatever you have in mind).

So how do you get all that on your computer in no time? The most kick-ass method is virtualenv, so let's look at that first.

virtualenv

Virtualenv is what you want to use during development and in production if you have shell access. So first: what does virtualenv do? If you are like me and you like Python, chances are you want to use it for another project as well. Now the more projects you have, the more likely it is that you will be working with different versions of Python itself or a library involved. Because let's face it: quite often libraries break backwards compatibility and it's unlikely that your application will not have any dependencies, that just won't happen. So virtualenv for the rescue!

It basically makes it possible to have multiple side-by-side "installations" of Python, each for your own project. It's not actually an installation but a clever way to keep things separated.

So let's see how that works!

If you are on OS X or Linux chances are that one of the following two commands will for for you:

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

Changes are you have virtualenv installed on your system then. Maybe it's even in your package manager (on ubuntu try `sudo apt-get install python-virtualenv`).

If you are on Windows and missing the *easy_install* command you have to install it first. Check the *easy_install on Windows* section for more information about how to do that.

So now that you have virtualenv running just fire up a shell and create your own environment. I usually create a folder and a *env* folder within:

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

Now you only have to activate it, whenever you work with it. On OS X and Linux do the following:

```
$ source env/bin/activate
```

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

```
$ env\scripts\activate
```

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

Now you can just enter the following command to get Flask activated in your virtualenv:

```
$ easy_install Flask
```

A few seconds later you are good to go.

System Wide Installation

This is possible as well, but I would not recommend it. Just run *easy_install* with root rights:

```
sudo easy_install Flask
```

(Run it in an Admin shell on Windows systems and without the *sudo*).

The Drop into Place Version

Now I really don't recommend this way on using Flask, but you can do that of course as well. Download the *dip* zipfile from the website and unzip it next to your application.

easy_install on Windows

On Windows installation of *easy_install* is a little bit tricker because on Windows slightly different rules apply, but it's not a biggy. The easiest way to accomplish that is downloading the ez_setup.py file and running it. (Double clicking should do the trick)

Once you have done that it's important to add the *easy_install* command and other Python scripts to the path. To do that you have to add the Python installation's Script folder to the *PATH* variable.

To do that, click right on your "Computer" desktop icon and click "Properties". On Windows Vista and Windows 7 then click on "Advanced System settings", on Windows XP click on the "Advanced" tab instead. Then click on the "Environment variables" button and double click on the "Path" variable in the "System variables" section.

There append the path of your Python interpreter's Script folder to the end of the last (make sure you delimit it from existing values with a semicolon). Assuming you are using Python 2.6 on the default path, add the following value:

```
;C:\Python26\Scripts
```

Then you are done. To check if it worked, open the cmd and execute "easy_install". If you have UAC enabled it should prompt you for admin privileges.

Quickstart

Eager to get started? This page gives a good introduction in how to gets started with Flask. This 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 that:

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

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

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

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

```
$ python hello.py
 * Running on http://localhost:5000/
```

Head over to http://localhost:5000/, you should see your hello world greeting.

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 it. We pass it the name of the module / package. This is needed so that Flask knows where it should look for templates, static files and so on.
3. Then we use the `route()` decorator to tell Flask what URL should trigger our function.
4. The function then has a name which is also used to generate URLs to that particular function, and returns the message we want to display in the user's browser.
5. Finally we use the `run()` function to run the local server with our application. The `if __name__ == '__main__':` makes sure the server only runs if the script is executed directly from the Python interpreter and not used as imported module.

To stop the server, hit control-C.

Troubleshooting

The browser is unable to access the server? Sometimes this is unfortunately caused by broken IPv6 support in your operating system, browser or a combination. For example on Snow Leopard Google Chrome is known to exhibit this behaviour.

If the browser does not load up the page, you can change the *app.run* call to force IPv4 usage:

```
if __name__ == '__main__':
    app.run(host='127.0.0.1')
```

Debug Mode

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

There are two ways to enable debugging. Either set that flag on the applciation object:

```
app.debug = True
app.run()
```

Or pass it to run:

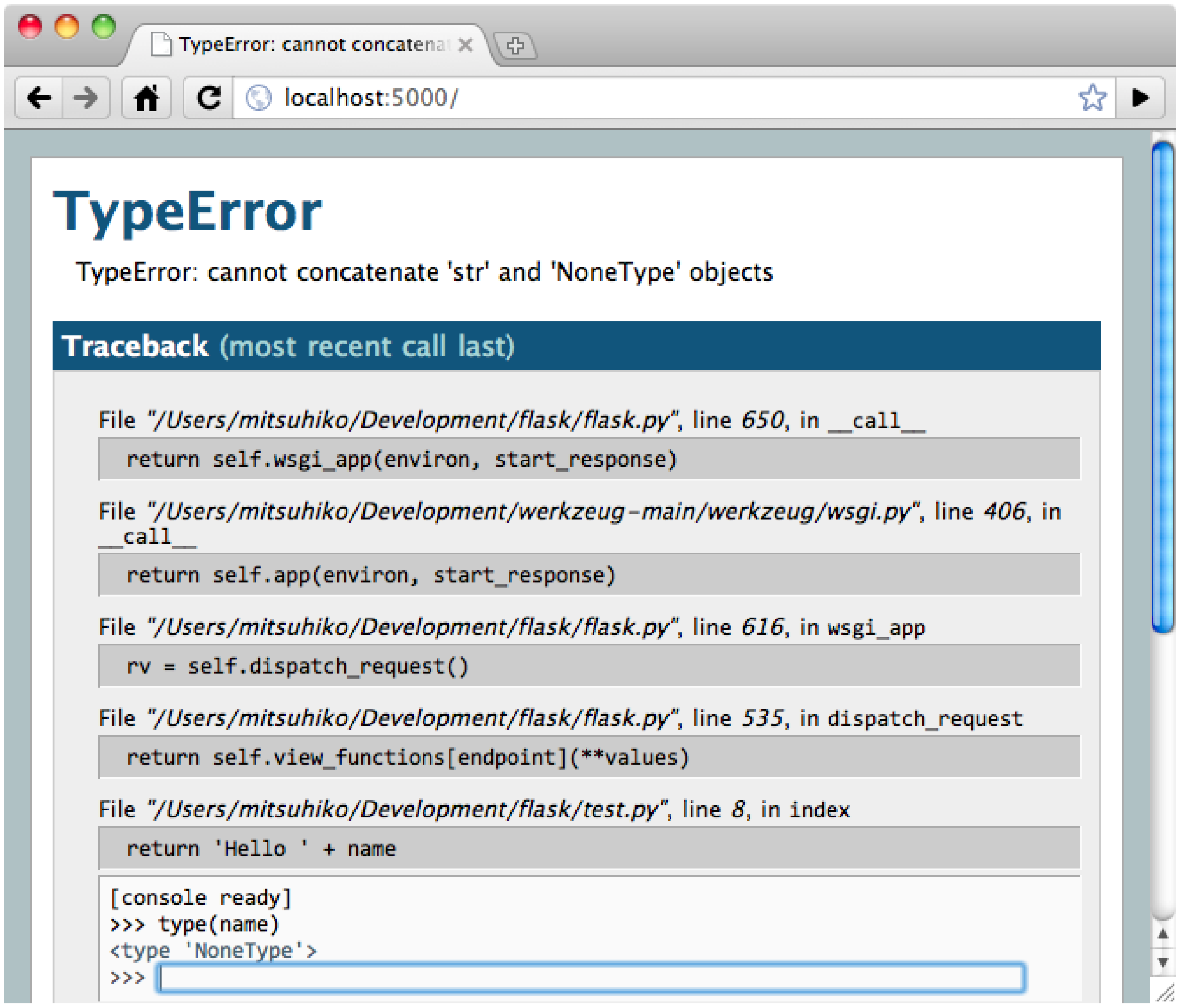
```
app.run(debug=True)
```

Both will have exactly the same effect.

Attention

The interactive debugger however does not work in forking environments which makes it nearly impossible to use on production servers but the debugger still allows the execution of arbitrary code which makes it a major security risk and **must never be used on production machines** because of that.

Screenshot of the debugger in action:



TypeError

TypeError: cannot concatenate 'str' and 'NoneType' objects

Traceback (most recent call last)

```
File "/Users/mitsuhiko/Development/flask/flask.py", line 650, in __call__
  return self.wsgi_app(environ, start_response)
File "/Users/mitsuhiko/Development/werkzeug-main/werkzeug/wsgi.py", line 406, in
__call__
  return self.app(environ, start_response)
File "/Users/mitsuhiko/Development/flask/flask.py", line 616, in wsgi_app
  rv = self.dispatch_request()
File "/Users/mitsuhiko/Development/flask/flask.py", line 535, in dispatch_request
  return self.view_functions[endpoint](**values)
File "/Users/mitsuhiko/Development/flask/test.py", line 8, in index
  return 'Hello ' + name
[console ready]
>>> type(name)
<type 'NoneType'>
>>>
```

Routing

As you have seen above, the `route()` decorator is used to bind a function to a URL. But there is more to it! You can make certain parts of the URL dynamic and attach multiple rules to a function.

Here some examples:

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

@app.route('/hello')
def hello():
    return 'Hello World'
```

Variable Rules

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 he will like the page and come back next time.

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

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

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

The following converters exist:

int	accepts integers
float	like *int* but for floating point values
path	like the default but also accepts slashes

URL Building

If it can match URLs, can it also generate them? Of course you 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. Here 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('profile', username='John Doe')
...
/
/login
/user/John%20Doe
```

(This also uses the *`test_request_context()`* method explained below. It basically tells flask to think we are handling a request even though we are not, we are in an interactive Python shell. Have a look at the explanation below. *Context Locals*).

Why would you want to build URLs instead of hardcoding them in your templates? There are three good reasons for this:

1. reversing is often more descriptive than hardcoding the URLs. Also and more importantly you can change URLs in one go without having to change the URLs all over the place.
2. URL building will handle escaping of special characters and unicode data transparently for you, you don't have to deal with that.
3. If your application is placed outside the URL root (so 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 to access 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 some examples:

```
@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 like the HTTP RFC (the document describing the HTTP protocol) demands, so you can completely ignore that part of the HTTP specification.

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

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

GET The Browser tells the server: just *get* me the information stored on that page and send them to me. This is probably the most common method.

HEAD The Browser tells the server: get me the information, but I am 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 not deliver the actual contents. 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 are usually transmitting 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 any useful, but there are some good reasons to do that. Consider the connection is lost during transmission, in that situation a system between the browser and the server might sent 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 that the given location.

Now the interesting part is that in HTML4 and XHTML1, the only methods a form might submit to the server are *GET* and *POST*. But with JavaScript and future HTML standards you can use other methods as well. Furthermore HTTP became quite popular lately and there are more things than browsers that are speaking HTTP. (Your revision control system for instance might speak HTTP)

Static Files

Dynamic web applications need static files as well. 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 to that part of the URL, use the special `'static'` URL 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 to provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here 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, that folder is next to that module, if it's a pacakge 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 Jinja2 Template Documentation for more information.

Here 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 (because for example 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 a basic introduction in 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'
```

Accessing Request Data

For web applications it's crucial to react to the data a client sent 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 are 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 just a standard global object, but actually a proxy to an object that is 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 webserver decides to spawn a new thread (or something else, the underlying object is capable of dealing with other concurrency systems than threads as well). 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 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 unittesting or something different. You will notice that code that 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 unittesting is by using 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 an example:

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

```
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`), but just mention some of the most common operations. First of all you have to import it from the 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 a full example of the two attributes mentioned above:

```
@app.route('/login', method=['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'
    # this is executed if the request method was GET or the
    # credentials were invalid
```

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('q', '')
```

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 a bit user unfriendly.

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

File Uploads

Obviously you can handle uploaded files with Flask just as easy. Just make sure not to forget to set the `enctype="multipart/form-data"` attribtue 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 a simple example 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 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))
    ...
```

Cookies

To access cookies you can use the *cookies* attribute. Again this 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.

Redirects and Errors

To redirect a user to somewhere else you can use the *redirect()* function, to abort a request early with an error code the *abort()* function. Here an example how this works:

```
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 he 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 translats to: all went well.

Sessions

Besides the request object there is also a second object called *session* that 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 signes the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless he knows 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 session, redirect, url_for, escape

@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 action="" 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 its there
    session.pop('username', None)

# set the secret key.  keep this really secret:
app.secret_key = 'the secret key'
```

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

Message Flashing

Good applications and user interfaces are all about feedback. If the user does not get enough feedback he 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.

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.

Tutorial

You want to develop an application with Python and Flask? Here you have the chance to learn that by example. In this tutorial we will create a simple microblog 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 database which comes out of the box with Python, so there is nothing else you need.

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

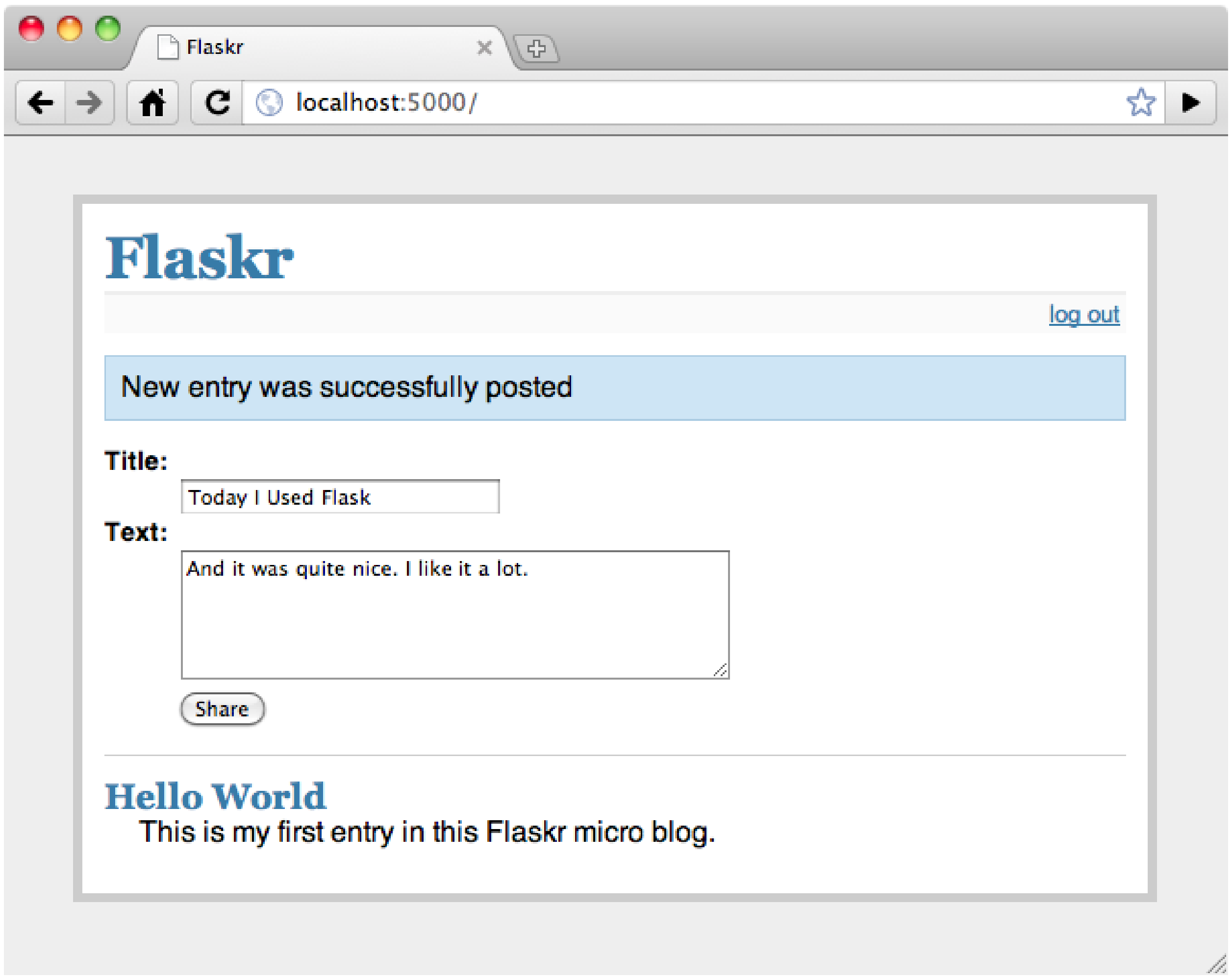
Introducing Flaskr

We will call our blogging application flaskr here, feel free to chose a less web-2.0-ish name ;) Basically we want it to 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 he or she 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 page shows all entries so far in reverse order (newest on top) and the user can add new ones from there if logged in.

We will be using SQlite3 directly for that application because it's good enough for an application of that size. For larger applications however it makes a lot of sense to use SQLAlchemy that handles database connections in a more intelligent way, allows 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 from the final application:



Flaskr

New entry was successfully posted

Title: Today I Used Flask

Text: And it was quite nice. I like it a lot.

Hello World

This is my first entry in this Flaskr micro blog.

Step 0: Creating The Folders

Before we get started, let's create the folders needed for this application:

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

The *flaskr* folder is not a python package, but just something where we drop our files. Directly into this folder we will then put our database schema as well as main module in the following steps. The files inside 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. Drop all the templates there.

Step 1: Database Schema

First we want to create the database schema. For this application only a single table is needed and we only want to support SQLite so that is quite easy. Just put the following contents into a file named *schema.sql* in the just created *flaskr* folder:

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

This schema consists of a single table called *entries* and 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.

Step 2: Application Setup Code

Now that we have the schema in place we can create the application module. Let's call it *flaskr.py* inside the *flaskr* folder. For starters we will add the imports we will need as well as the config section. For small applications it's a possibility to drop the configuration directly into the module which we will be doing here. However a cleaner solution would be to create a separate *.ini* or *.py* file and load that or import the values from there.

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

# configuration
DATABASE = '/tmp/flaskr.db'
DEBUG = True
SECRET_KEY = 'development key'
USERNAME = 'admin'
PASSWORD = 'default'
```

Next we can create our actual application and initialize it with the config:

```
# create our little application :)
app = Flask(__name__)
app.secret_key = SECRET_KEY
app.debug = DEBUG
```

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. The debug flag enables or disables the interactive debugger. Never leave debug mode activated in a production system because it will allow users to executed code on the server!

We also add a method to easily connect to the database specified. That 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

```
def connect_db():
    return sqlite3.connect(DATABASE)
```

Finally we just add a line to the bottom of the file that fires up the server if we run that file as standalone application:

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

With that out of the way you should be able to start up the application without problems. When you head over to the server you will get an 404 page not found error because we don't have any views yet. But we will focus on that a little later. First we should get the database working.

Troubleshooting

If you notice later that the browser cannot connect to the server during development, you might want to try this line instead:

```
app.run(host='127.0.0.1')
```

In a nutshell: Werkzeug starts up as IPv6 on many operating systems by default and not every browser is happy with that. This forces IPv4 usage.

Step 3: Creating The Database

Flaskr is a database powered application as outlined earlier, and more precisely, an application powered by a relational database system. Such systems need a schema that tells them how to store that information. So 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. Also one has to provide the path to the database there which leaves some place for errors. It's a good idea to add a function that initializes the database for you to the application.

If you want to do that, you first have to import the `contextlib.closing()` function from the contextlib package. If you want to use Python 2.5 it's also necessary to enable the *with* statement first (*__future__* imports must be the very first import):

```
from __future__ import with_statement
from contextlib import closing
```

Next we can create a function called *init_db* that initializes the database. For this we can use the *connect_db* function we defined earlier. Just add that function below the *connect_db* function:

```
def init_db():
    with closing(connect_db()) as db:
        with app.open_resource('schema.sql') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

The `closing()` helper function allows us to keep a connection open for the duration of the *with* block. The `open_resource()` method of the application object supports that functionality out of the box, so it can be used in the *with* block directly. This function opens a file from the resource location (your *flaskr* folder) and allows you to read from it. We are using this here to execute a script on the database connection.

When we connect to a database we get a connection object (here called *db*) that can give us a cursor. On that cursor there is a method to execute a complete script. Finally we only have to commit the changes. SQLite 3 and other transactional databases will not commit unless you explicitly tell it to.

Now it is possible to create a database by starting up a Python shell and importing and calling that function:

```
>>> from flaskr import init_db
>>> init_db()
```

Step 4: Request Database Connections

Now we know how we can open database connections and use them for scripts, but how can we elegantly do that for requests? We will need the database connection in all our functions so it makes sense to initialize them before each request and shut them down afterwards.

Flask allows us to do that with the `before_request()` and `after_request()` decorators:

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

@app.after_request
def after_request(response):
    g.db.close()
    return response
```

Functions marked with `before_request()` are called before a request and passed no arguments, functions marked with `after_request()` are called after a request and passed the response that will be sent to the client. They have to return that response object or a different one. In this case we just return it unchanged.

We store our current database connection on the special `g` object that flask provides for us. This object stores information for one request only and is available from within each function. Never store such things on other objects because this would not work with threaded environments. That special `g` object does some magic behind the scenes to ensure it does the right thing.

Step 5: The View Functions

Now that the database connections are working we can start writing the view functions. We 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) on top. The rows returned from the cursor are tuples with the columns ordered like specified in the select statement. This is good enough for small applications like here, but you might want to convert them into a dict. If you are interested how to do that, check out the *Easy Querying* example.

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

```
@app.route('/')
def show_entries():
    cur = g.db.execute('select title, text from entries order by id desc')
    entries = [dict(title=row[0], text=row[1]) for row in cur.fetchall()]
    return render_template('show_entries.html', entries=entries)
```

Add New Entry

This view lets the user add new entries if he's logged in. This only responds to *POST* requests, the actual form is shown on the *show_entries* page. If everything worked out well we 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)
    g.db.execute('insert into entries (title, text) values (?, ?)',
                 [request.form['title'], request.form['text']])
    g.db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

Note that we check that the user is logged in here (the *logged_in* key is present in the session and *True*).

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 in 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 that case also a message is flashed that informs the user he or she was logged in successfully. If an error occoured the template is notified about that and the user asked again:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != USERNAME:
            error = 'Invalid username'
        elif request.form['password'] != 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. We use 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 was not in there. This is helpful because we don't have to check in that case 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'))
```

Step 6: The Templates

Now we should start working on the templates. If we request the URLs now we would only 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. We also tell the form to submit to your *add_entry* 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

Finally 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 %}
```

Step 7: 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 we created before:

```
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; }
```

Bonus: Testing the Application

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

Testing Flask Applications

Something that is untested is broken.

Not sure where that is coming from, and it's not entirely correct, but also not that 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 however has automated tests you can savely change things and you will instantly know if your change broke something.

Flask gives you a couple of ways to test applications. It mainly does that by exposing the Werkzeug test `Client` class to your code and handling the context locals for you. You can then use that with your favourite testing solution. In this documentation we will us the `unittest` package that comes preinstalled with each Python installation.

The Application

First we need an application to test for functionality. For the testing 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 that, we add a second module (*flaskr_tests.py*) and create a unittest skeleton there:

```
import unittest
import flaskr
import tempfile

class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db = tempfile.NamedTemporaryFile()
        self.app = flaskr.app.test_client()
        flaskr.DATABASE = self.db.name
        flaskr.init_db()

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

The code in the *setUp* function creates a new test client and initialize a new database. That function is called before each individual test function. What the test client does for us is giving 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. Just make sure that you keep a reference to the `NamedTemporaryFile` around (we store it as *self.db* because of that) so that the garbage collector does not remove that object and with it the database from the filesystem.

If we now run that testsuite, we should see the following output:

```
$ python flaskr_tests.py

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

OK
```

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

The First Test

Now we can add the first test. Let's check that the application shows "No entries here so far" if we access the root of the application (`/`). For that we modify our created test case class so that it looks like this:

```
class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db = tempfile.NamedTemporaryFile()
        self.app = flaskr.app.test_client()
        flaskr.DATABASE = self.db.name
        flaskr.init_db()

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

Test functions begin with the word *test*. Every function named like that will be picked up automatically. 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
```

Of course you can submit forms with the test client as well which we will use now to log our user in.

Logging In and Out

The majority of the functionality of our application is only available for the administration user. So we need a way to log our test client into the application and out of it again. For that we fire some requests to the login and logout pages with the required form data (username and password). Because the login and logout pages redirect, we tell the client to *follow_redirects*.

Add the following two methods do 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 if logging in and out works and that it fails with invalid credentials. Add this as new test to the class:

```
def test_login_logout(self):
    rv = self.login(flaskr.USERNAME, flaskr.PASSWORD)
    assert 'You were logged in' in rv.data
    rv = self.logout()
    assert 'You were logged out' in rv.data
    rv = self.login(flaskr.USERNAME + 'x', flaskr.PASSWORD)
    assert 'Invalid username' in rv.data
    rv = self.login(flaskr.USERNAME, flaskr.PASSWORD + 'x')
    assert 'Invalid password' in rv.data
```

Test Adding Messages

Now we can also test that adding messages works. Add a new test method like this:

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

Here we also 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. That one contains a larger test suite.

Patterns for Flask

Certain things are common enough that the changes 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, changes 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.

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
        ...
```

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 *runserver.py* next to the inner *yourapplication* folder with the following contents:

```
from yourapplication import app
app.run(debug=True)
```

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 have to be in the *__init__.py* file. That way each module can import it safely and the *__name__* variable will resole to the correct package.
2. all the view functions (the ones with a `route()` decorator on top) have to be imported when in the *__init__.py* file. Not the objects itself, but the module it is in. Do the importing at the *bottom* of the file.

Here 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!'
```

Circular Imports

Every Python programmer hates it, and yet we just did that: circular imports (That's when two module depend on each one. 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.

Using SQLite 3 with Flask

In Flask you can implement opening of dabase connections at the beginning of the request and closing at the end with the `before_request()` and `after_request()` decorators in combination with the special `g` object.

So here a simple example how you can use SQLite 3 with Flask:

```
import sqlite3
from flask import g

DATABASE = '/path/to/database.db'

def connect_db():
    return sqlite3.connect(DATABASE)

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

@app.after_request
def after_request(response):
    g.db.close()
    return response
```

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 helper function can be useful:

```
def query_db(query, args=(), one=False):
    cur = g.db.execute(query, args)
    rv = [dict((cur.description[idx][0], value)
               for idx, value in enumerate(row)) for row in cur.fetchall()]
    return (rv[0] if rv else None) if one else rv
```

This handy little function 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 formattings 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 bases on that schema. This function can do that for you:

```
from contextlib import closing

def init_db():
    with closing(connect_db()) as db:
        with app.open_resource('schema.sql') 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*). Although that is not necessary but makes a lot of sense.

There are three very common ways to use SQLAlchemy. I will outline each of them here:

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 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')
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    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 for you:

```
from yourapplication.database import db_session

@app.after_request
def shutdown_session(response):
    db_session.remove()
    return response
```

Here an example model (put that into *models.py* for instance):

```
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, self.email)
```

You can insert entries into the database like this then:

```
>>> 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

coming soon

SQL Abstraction Layer

coming soon

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>
```

In this example, the `{% block %}` tags define four blocks that child templates can fill in. All the *block* tag does is to 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 he 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.

So here a full example:

```
from flask import flash, redirect, url_for, render_template

@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 sucessfully logged in')
            return redirect(url_for('index'))
    return render_template('login.html', error=error)
```

And here 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 %}
```

And here the index.html template:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Overview</h1>
  <p>Do you want to <a href="{{ url_for('login') }}">log in?</a>
{% endblock %}
```

And of course the login template:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Login</h1>
  {% if error %}
    <p class=error><strong>Error:</strong> {{ error }}
  {% endif %}
  <form action="" 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 %}
```

Deployment Options

Depending on what you have available there are multiple ways to run Flask applications. A very common method is to use the builtin server during development and maybe behind a proxy for simple applications, but there are more options available.

If you have a different WSGI server look up the server documentation about how to use a WSGI app with it. Just remember that your application object is the actual WSGI application.

FastCGI

A very popular deployment setup on servers like lighttpd and nginx is FastCGI. 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.

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

WSGIServer(app).run()
```

This is enough for Apache to work, however lighttpd and nginx need a socket 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 lighttpd

A basic FastCGI configuration for lighttpd looks like that:

```
fastcgi.server = ("/yourapplication" =>
    "yourapplication" => (
        "socket" => "/tmp/yourapplication-fcgi.sock",
        "bin-path" => "/var/www/yourapplication/yourapplication.fcgi",
        "check-local" => "disable"
    )
)
```

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.

Configuring nginx

Installing FastCGI applications on nginx is a bit tricky because by default some FastCGI parameters are not properly forwarded.

A basic FastCGI configuration for nginx looks like this:

```
location /yourapplication/ {
    include fastcgi_params;
    if ($uri ~ ^/yourapplication/(.*)?) {
        set $path_url $1;
    }
    fastcgi_param PATH_INFO $path_url;
    fastcgi_param SCRIPT_NAME /yourapplication;
    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 easier because you don't have to figure out how to calculate *PATH_INFO* and *SCRIPT_NAME*:

```
location /yourapplication/ {
    include fastcgi_params;
    fastcgi_param PATH_INFO $fastcgi_script_name;
    fastcgi_param SCRIPT_NAME "";
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

Since Nginx doesn't load FastCGI apps, you have to do it by yourself. You can either write an *init.d* script for that or execute it inside a screen session:

```
$ screen
$ /var/www/yourapplication/yourapplication.fcgi
```

Debugging

FastCGI deployments tend to be hard to debug on most webservers. 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 webserver user is *www-data*:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcg", 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.

mod_wsgi (Apache)

If you are using the Apache webserver you should consider using 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 installation instructions for 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
```

On FreeBSD install *mod_wsgi* by compiling the *www/mod_wsgi* port or by using pkg_add:

```
# pkg_add -r mod_wsgi
```

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 where you will find it again (eg: */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.

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>
```

For more information consult the mod_wsgi wiki.

Tornado

Tornado is an open source version of the scalable, non-blocking web server and tools that power FriendFeed. Because it is non-blocking and uses epoll, it can handle thousands of simultaneous standing connections, which means it is ideal for real-time web services. Integrating this service with Flask is a trivial task:

```
from tornado.wsgi import WSGIContainer
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from yourapplication import app

http_server = HTTPServer(WSGIContainer(app))
http_server.listen(5000)
IOLoop.instance().start()
```

Gevent

Gevent is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of libevent event loop:

```
from gevent.wsgi import WSGIServer
from yourapplication import app

http_server = WSGIServer(('', 5000), app)
http_server.serve_forever()
```

CGI

If all other deployment methods do not work, CGI will work for sure. CGI is supported by all major browsers but usually has a less-than-optimal performance.

This is also the way you can use a Flask application on Google's AppEngine, there however the execution does happen in a CGI-like environment. The application's performance is unaffected because of that.

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)
```

If you're running Python 2.4 you will need the `wsgiref` package. Python 2.5 and higher ship this as part of the standard library.

Server Setup

Usually there are two ways to configure the server. Either just copy the *.cgi* into a *cgi-bin* (and use *mod_rerwite* or something similar to rewrite the URL) or let the server point to the file directly.

In Apache for example you can put a like like this into the config:

```
ScriptName /app /path/to/the/application.cgi
```

For more information consult the documentation of your webserver.

Becoming Big

Your application is becoming more and more complex? Flask is really not designed for large scale applications and does not attempt to do so, but that does not mean you picked the wrong tool in the first place.

Flask is powered by Werkzeug and Jinja2, two libraries that are in use at a number of large websites out there and all Flask does is bringing those two together. Being a microframework, Flask is literally a single file. What that means for large applications is that it's probably a good idea to take the code from Flask and put it into a new module within the applications and expanding on that.

What Could Be Improved?

For instance it makes a lot of sense to change the way endpoints (the names of the functions / URL rules) are handled to also take the module name into account. Right now the function name is the URL name, but imagine you have a large applications consisting of multiple components. In that case, it makes a lot of sense to use dotted names for the URL endpoints.

Here some suggestions how Flask can be modified to better accomodate large scale applications:

- implement dotted names for URL endpoints
- get rid of the decorator function registering which causes a lot of troubles for applications that have circular dependencies. It also requires that the whole application is imported when the system initializes or certain URLs will not be available right away. A better solution would be to have one module with all URLs in there and specifing the target functions explicitly or by name and importing them when needed.
- switch to explicit request object passing. This makes it more to type (because you now have something to pass around) but it makes it a whole lot easier to debug hairy situations and to test the code.
- integrate the Babel i18n package or SQLAlchemy directly into the core framework.

Why does not Flask do all that by Default?

There is a huge difference between a small application that only has to handle a couple of requests per second and with an overall code complexity of less than 4000 lines of code or something of larger scale. At one point it becomes important to integrate external systems, different storage backends and more.

If Flask was designed with all these contingencies in mind, it would be a much more complex framework and less easy to get started with.

CHAPTER 2

Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you:

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`**(*package_name*)

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__)
```

`add_url_rule`(*rule*, *endpoint*, ***options*)

Connects a URL rule. Works exactly like the `route()` decorator but does not register the view function for the endpoint.

Basically this example:

```
@app.route('/')
def index():
    pass
```

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('index', '/')
app.view_functions['index'] = index
```

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

after_request(*f*)
Register a function to be run after each request.

after_request_funcs = None
a list of functions that are called at the end of the request. Tha function is passed the current response object and modify it in place or replace it. To register a function here use the `after_request()` decorator.

before_request(*f*)
Registers a function to run before each request.

before_request_funcs = None
a list of functions that should be called at the beginning of the request before request dispatching kicks in. 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.

context_processor(*f*)
Registers a template context processor function.

create_jinja_loader()
Creates the Jinja loader. By default just a package loader for the configured package is returned that looks up templates in the *templates* folder. To add other loaders it's possible to override this method.

debug = None
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 ocurrs and the integrated server will automatically reload the application if changes in the code are detected.

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()`.

error_handlers = None
a dictionary of all registered error handlers. The key is be the error code as integer, the value the function that should handle that error. To register a error handler, use the `errorhandler()` decorator.

errorhandler(*code*)
A decorator that is used to register a function give a given error code. Example:

```
@app.errorhandler(404)
def page_not_found():
    return 'This page does not exist', 404
```

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():
    return 'This page does not exist', 404
app.error_handlers[404] = page_not_found
```

Parameters **code** – the code as integer for the handler

jinja_env = None
the Jinja2 environment. It is created from the `jinja_options` and the loader that is returned by the `create_jinja_loader()` function.

jinja_options = {'autoescape': True, 'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.with_']}
options that are passed directly to the Jinja2 environment

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 allowd 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
`tuple`	the response object is created with the contents of the tuple as arguments
a WSGI function	the function is called as WSGI application and buffered as response object

Parameters **rv** – the return value from the view function

match_request()
Matches the current request against the URL map and also stores the endpoint and view arguments on the request object is successful, otherwise the exception is stored.

open_resource(*resource*)
Opens a resource from the application's resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schemal.sql
/static
    /style.css
/template
    /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.

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.

Parameters **request** – an instance of *request_class*.

package_name = **None**
the name of the package or module. Do not change this once it was set by the constructor.

preprocess_request()
Called before the actual request dispatching and will call every as *before_request()* decorated function. If any of these function returns a value it's handled as if it was the return value from the view and further request handling is stopped.

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.

Parameters **response** – a *response_class* object.

Returns a new response object or the same, has to be an instance of *response_class*.

request_class
the class that is used for request objects. See *request* for more information.

alias of *Request*

request_context(*environ*)
Creates a request context 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)
```

Params environ a WSGI environment

response_class
the class that is used for response objects. See *Response* for more information.

alias of *Response*

root_path = **None**
where is the app root located?

route(*rule*, ***options*)
A decorator that is used to register a view function for a given URL rule. Example:

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

Variables 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 possible:

int	accepts integers
float	like *int* but for floating point values
path	like the default but also accepts slashes

Here 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 request 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.

The `route()` decorator accepts a couple of other arguments as well:

Parameters

- **`rule`** – the URL rule as string
- **`methods`** – a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`).
- **`subdomain`** – specifies the rule for the subdoain in case subdomain matching is in use.
- **`strict_slashes`** – can be used to disable the strict slashes setting for this rule. See above.
- **`options`** – other options to be forwarded to the underlying `Rule` object.

`run`(*host='localhost'*, *port=5000*, ***options*)

Runs the application on a local development server. If the `debug` flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

Parameters

- **`host`** – the hostname to listen on. set this to `'0.0.0.0'` to have the server available externally as well.
- **`port`** – the port of the webserver
- **`options`** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.run_simple()` for more information.

`save_session`(*session*, *response*)
Saves the session if it needs updates. For the default implementation, check *`open_session()`*.

Parameters

- **`session`** – the session to be saved (a `SecureCookie` object)
- **`response`** – an instance of *`response_class`*

`secret_key` **= None**
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.

`session_cookie_name` **= 'session'**
The secure cookie uses this for the name of the session cookie

`static_path` **= '/static'**
path for the static files. If you don't want to use static files you can set this value to *None* in which case no URL rule is added and the development server will no longer serve any static files.

`template_context_processors` **= None**
a list of functions that are called without arguments to populate the template context. Each returns a dictionary that the template context is updated with. To register a function here, use the *`context_processor()`* decorator.

`test_client`()
Creates a test client for this application. For information about unit testing head over to *Testing Flask Applications*.

`test_request_context`(**args*, ***kwargs*)
Creates a WSGI environment from the given values (see `werkzeug.create_environ()` for more information, this function accepts the same arguments).

`update_template_context`(*context*)
Update the template context with some commonly used variables. This injects request, session and g into the template context.

Parameters **`context`** – the context as a dictionary that is updated in place to add extra variables.

`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:

app.wsgi_app = MyMiddleware(app.wsgi_app)

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

Incoming Request Data

class `flask.`**`Request`**(*environ*)
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.

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.

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.

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 encoding (for example it was transmitted as JSON) the data is stored unmodified in this stream for consumption. For example to read the incoming request data as JSON, one can do the following:

```
json_body = simplejson.load(request.stream)
```

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

script_root

url

base_url

url_root

Provides different ways to look at the current URL. Imagine your application is listening on the following URL:

```
http://www.example.com/myapplication
```

And a user requests the following URL:

```
http://www.example.com/myapplication/page.html?x=y
```

In this case the values of the above mentioned attributes would be the following:

path	`/page.html`
script_root	`/myapplication`
url	`http://www.example.com/myapplication/page.html`
base_url	`http://www.example.com/myapplication/page.html?x=y`
root_url	`http://www.example.com/myapplication/`

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 a 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 `request_class` to your subclass.

`headers`

A `Headers` object representing the response headers.

`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 he knows 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.

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
```

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.

Useful Functions and Classes

`flask.`**`url_for`**(*endpoint*, ***values*)

Generates a URL to the given endpoint with the method provided.

Parameters

- **`endpoint`** – the endpoint of the URL (name of the function)
- **`values`** – the variable arguments of the URL rule

`flask.`**`abort`**(*code*)

Raises an `HTTPException` for the given status code. For example to abort request handling with a page not found exception, you would call `abort(404)`.

Parameters **`code`** – the HTTP error code.

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.**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*)
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()`*.

Parameters **`message`** – the message to be flashed.

`flask.`**`get_flashed_messages`**()
Pulls all flashed messages from the session and returns them. Further calls in the same request to the function will return the same messages.

Template Rendering

`flask.`**`render_template`**(*template_name*, ***context*)
Renders a template from the template folder with the given context.

Parameters

- **`template_name`** – the name of the template to 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.

Parameters

- **`template_name`** – the sourcecode of the template to be rendered
- **`context`** – the variables that should be available in the context of the template.

Python Module Index

f

flask, 39

Index

A

abort() (in module flask), 47
add_url_rule() (flask.Flask method), 39
after_request() (flask.Flask method), 40
after_request_funcs (flask.Flask attribute), 40
args (flask.request attribute), 45

B

base_url (flask.request attribute), 45
before_request() (flask.Flask method), 40
before_request_funcs (flask.Flask attribute), 40

C

context_processor() (flask.Flask method), 40
cookies (flask.request attribute), 45
create_jinja_loader() (flask.Flask method), 40

D

data (flask.Response attribute), 46
debug (flask.Flask attribute), 40
dispatch_request() (flask.Flask method), 40

E

environ (flask.request attribute), 45
error_handlers (flask.Flask attribute), 40
errorhandler() (flask.Flask method), 40
escape() (flask.Markup class method), 49
escape() (in module flask), 48

F

files (flask.request attribute), 45
flash() (in module flask), 49
Flask (class in flask), 39
flask (module), 39
form (flask.request attribute), 45

G

g (in module flask), 47
get_flashed_messages() (in module flask), 49

H

headers (flask.Response attribute), 46

J

jinja_env (flask.Flask attribute), 41
jinja_options (flask.Flask attribute), 41

M

make_response() (flask.Flask method), 41
Markup (class in flask), 48
match_request() (flask.Flask method), 41
method (flask.request attribute), 45
mimetype (flask.Response attribute), 46
modified (flask.session attribute), 47

N

new (flask.session attribute), 47

O

open_resource() (flask.Flask method), 41
open_session() (flask.Flask method), 42

P

package_name (flask.Flask attribute), 42
path (flask.request attribute), 45
preprocess_request() (flask.Flask method), 42
process_response() (flask.Flask method), 42

R

redirect() (in module flask), 47
render_template() (in module flask), 49
render_template_string() (in module flask), 49
Request (class in flask), 44
request (class in flask), 45
request_class (flask.Flask attribute), 42
request_context() (flask.Flask method), 42
Response (class in flask), 46
response_class (flask.Flask attribute), 42
root_path (flask.Flask attribute), 42

route() (flask.Flask method), 42
run() (flask.Flask method), 43

S

save_session() (flask.Flask method), 43
script_root (flask.request attribute), 45
secret_key (flask.Flask attribute), 44
session (class in flask), 47
session_cookie_name (flask.Flask attribute), 44
set_cookie() (flask.Response method), 46
static_path (flask.Flask attribute), 44
status_code (flask.Response attribute), 46
stream (flask.request attribute), 45
striptags() (flask.Markup method), 49

T

template_context_processors (flask.Flask attribute), 44
test_client() (flask.Flask method), 44
test_request_context() (flask.Flask method), 44

U

unescape() (flask.Markup method), 49
update_template_context() (flask.Flask method), 44
url (flask.request attribute), 45
url_for() (in module flask), 47
url_root (flask.request attribute), 45

V

values (flask.request attribute), 45
view_functions (flask.Flask attribute), 44

W

wsgi_app() (flask.Flask method), 44