



Flask Documentation

Release 0.8-dev

August 09, 2011

CONTENTS

I	User's Guide	1
1	Foreword	3
1.1	What does “micro” mean?	3
1.2	A Framework and an Example	4
1.3	Web Development is Dangerous	4
1.4	The Status of Python 3	4
2	Installation	7
2.1	virtualenv	7
2.2	System Wide Installation	8
2.3	Living on the Edge	9
2.4	<i>easy_install</i> on Windows	9
3	Quickstart	11
3.1	A Minimal Application	11
3.2	Debug Mode	12
3.3	Routing	13
3.4	Static Files	17
3.5	Rendering Templates	17
3.6	Accessing Request Data	19
3.7	Redirects and Errors	22
3.8	About Responses	23
3.9	Sessions	24
3.10	Message Flashing	25
3.11	Logging	25
3.12	Hooking in WSGI Middlewares	25
4	Tutorial	27
4.1	Introducing Flaskr	27
4.2	Step 0: Creating The Folders	28
4.3	Step 1: Database Schema	29
4.4	Step 2: Application Setup Code	29
4.5	Step 3: Creating The Database	31
4.6	Step 4: Request Database Connections	32
4.7	Step 5: The View Functions	33
4.8	Step 6: The Templates	34

4.9	Step 7: Adding Style	36
4.10	Bonus: Testing the Application	37
5	Templates	39
5.1	Jinja Setup	39
5.2	Standard Context	39
5.3	Standard Filters	40
5.4	Controlling Autoescaping	40
5.5	Registering Filters	41
5.6	Context Processors	42
6	Testing Flask Applications	43
6.1	The Application	43
6.2	The Testing Skeleton	43
6.3	The First Test	44
6.4	Logging In and Out	45
6.5	Test Adding Messages	46
6.6	Other Testing Tricks	46
6.7	Keeping the Context Around	47
7	Handling Application Errors	49
7.1	Error Mails	49
7.2	Logging to a File	50
7.3	Controlling the Log Format	51
7.4	Other Libraries	52
8	Configuration Handling	55
8.1	Configuration Basics	55
8.2	Builtin Configuration Values	56
8.3	Configuring from Files	57
8.4	Configuration Best Practices	58
8.5	Development / Production	58
9	Signals	61
9.1	Subscribing to Signals	61
9.2	Creating Signals	63
9.3	Sending Signals	63
9.4	Decorator Based Signal Subscriptions	64
9.5	Core Signals	64
10	Pluggable Views	67
10.1	Basic Principle	67
10.2	Method Hints	68
10.3	Method Based Dispatching	69
11	The Request Context	71
11.1	Diving into Context Locals	71
11.2	How the Context Works	72
11.3	Callbacks and Errors	73

11.4	Teardown Callbacks	73
11.5	Notes On Proxies	74
11.6	Context Preservation on Error	74
12	Modular Applications with Blueprints	77
12.1	Why Blueprints?	77
12.2	The Concept of Blueprints	78
12.3	My First Blueprint	78
12.4	Registering Blueprints	78
12.5	Blueprint Resources	79
12.6	Building URLs	80
13	Working with the Shell	83
13.1	Creating a Request Context	83
13.2	Firing Before/After Request	84
13.3	Further Improving the Shell Experience	84
14	Patterns for Flask	85
14.1	Larger Applications	85
14.2	Application Factories	87
14.3	Application Dispatching	89
14.4	Using URL Processors	92
14.5	Deploying with Distribute	94
14.6	Deploying with Fabric	97
14.7	Using SQLite 3 with Flask	101
14.8	SQLAlchemy in Flask	103
14.9	Uploading Files	106
14.10	Caching	110
14.11	View Decorators	111
14.12	Form Validation with WTForms	114
14.13	Template Inheritance	116
14.14	Message Flashing	117
14.15	AJAX with jQuery	119
14.16	Custom Error Pages	122
14.17	Lazily Loading Views	123
14.18	MongoKit in Flask	125
14.19	Adding a favicon	128
14.20	Streaming Contents	129
15	Deployment Options	131
15.1	mod_wsgi (Apache)	131
15.2	CGI	134
15.3	FastCGI	135
15.4	uWSGI	137
15.5	Other Servers	138
16	Becoming Big	141
16.1	Why consider Forking?	141
16.2	Scaling like a Pro	142

16.3	Dialogue with the Community	142
II	API Reference	143
17	API	145
17.1	Application Object	145
17.2	Blueprint Objects	160
17.3	Incoming Request Data	163
17.4	Response Objects	165
17.5	Sessions	166
17.6	Session Interface	167
17.7	Application Globals	169
17.8	Useful Functions and Classes	169
17.9	Message Flashing	175
17.10	Returning JSON	175
17.11	Template Rendering	176
17.12	Configuration	177
17.13	Useful Internals	179
17.14	Signals	181
17.15	Class Based Views	182
17.16	View Function Options	183
III	Additional Notes	185
18	Design Decisions in Flask	187
18.1	The Explicit Application Object	187
18.2	One Template Engine	188
18.3	Micro with Dependencies	189
18.4	Thread Locals	189
18.5	What Flask is, What Flask is Not	190
19	HTML/XHTML FAQ	191
19.1	History of XHTML	191
19.2	History of HTML5	192
19.3	HTML versus XHTML	192
19.4	What does “strict” mean?	193
19.5	New technologies in HTML5	194
19.6	What should be used?	194
20	Security Considerations	195
20.1	Cross-Site Scripting (XSS)	195
20.2	Cross-Site Request Forgery (CSRF)	196
20.3	JSON Security	196
21	Unicode in Flask	199
21.1	Automatic Conversion	199
21.2	The Golden Rule	200

21.3	Encoding and Decoding Yourself	200
21.4	Configuring Editors	200
22	Flask Extension Development	203
22.1	Anatomy of an Extension	203
22.2	“Hello Flaskext!”	204
22.3	Initializing Extensions	205
22.4	The Extension Code	206
22.5	Adding an <i>init_app</i> Function	207
22.6	End-Of-Request Behavior	208
22.7	Learn from Others	209
22.8	Approved Extensions	209
23	Pocoo Styleguide	211
23.1	General Layout	211
23.2	Expressions and Statements	212
23.3	Naming Conventions	213
23.4	Docstrings	214
23.5	Comments	214
24	Upgrading to Newer Releases	215
24.1	Version 0.8	215
24.2	Version 0.7	216
24.3	Version 0.6	219
24.4	Version 0.5	219
24.5	Version 0.4	220
24.6	Version 0.3	220
25	Flask Changelog	223
25.1	Version 0.8	223
25.2	Version 0.7.3	223
25.3	Version 0.7.2	224
25.4	Version 0.7.1	224
25.5	Version 0.7	224
25.6	Version 0.6.1	225
25.7	Version 0.6	225
25.8	Version 0.5.2	226
25.9	Version 0.5.1	226
25.10	Version 0.5	226
25.11	Version 0.4	227
25.12	Version 0.3.1	227
25.13	Version 0.3	228
25.14	Version 0.2	228
25.15	Version 0.1	228
26	License	229
26.1	Authors	229
26.2	General License Definitions	230
26.3	Flask License	230

26.4	Flask Artwork License	231
	Index	233

Part I

USER'S GUIDE

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

FOREWORD

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

1.1 What does “micro” mean?

To me, the “micro” in microframework refers not only to the simplicity and small size of the framework, but also to the typically limited complexity and size of applications that are written with the framework. Also the fact that you can have an entire application in a single Python file. To be approachable and concise, a microframework sacrifices a few features that may be necessary in larger or more complex applications.

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 might also cause some troubles for very large applications because changes on these thread-local objects can happen anywhere in the same thread.

Flask provides some tools to deal with the downsides of this approach but it might be an issue for larger applications because in theory modifications on these objects might happen anywhere in the same thread.

Flask is also based on convention over configuration, which means that many things are preconfigured. For example, by convention, templates and static files are in subdirectories within the Python source tree of the application.

The main reason however why Flask is called a “microframework” is the idea to keep the core simple but extensible. There is no database abstraction layer, no form validation or anything else where different libraries already exist that can handle that. However Flask knows the concept of extensions that can add this functionality into your application as if it was implemented in Flask itself. There are currently extensions for object relational mappers, form validation, upload handling, various open authentication technologies and more.

However Flask is not much code and it is built on a very solid foundation and with that it is very easy to adapt for large applications. If you are interested in that, check

out the *Becoming Big* chapter.

If you are curious about the Flask design principles, head over to the section about *Design Decisions in Flask*.

1.2 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 everything on its own; it reuses existing code.

1.3 Web Development is Dangerous

I'm not 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 securely.

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

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

So always keep security in mind when doing web development.

1.4 The Status of Python 3

Currently the Python community is in the process of improving libraries to support the new iteration of the Python programming language. While the situation is greatly improving there are still some issues that make it hard for us to switch over to Python 3 just now. These problems are partially caused by changes in the language that went unreviewed for too long, partially also because we have not quite worked out how the lower level API should change for the unicode differences in Python3.

Werkzeug and Flask will be ported to Python 3 as soon as a solution for the changes is found, and we will provide helpful tips how to upgrade existing applications to Python 3. Until then, we strongly recommend using Python 2.6 and 2.7 with activated Python 3 warnings during development. If you plan on upgrading to Python 3 in the near future we strongly recommend that you read [How to write forwards compatible Python code](#).

INSTALLATION

Flask depends on two external libraries, [Werkzeug](#) and [Jinja2](#). Werkzeug is a toolkit for WSGI, the standard Python interface between web applications and a variety of servers for both development and deployment. Jinja2 renders templates.

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

Either way, you will need Python 2.5 or higher to get started, so be sure to have an up to date Python 2.x installation. At the time of writing, the WSGI specification is not yet finalized for Python 3, so Flask cannot support the 3.x series of Python.

2.1 virtualenv

Virtualenv is probably what you want to use during development, and in production too if you have shell access there.

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

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

So let's see how virtualenv works!

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

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

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

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

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

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

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

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

```
$ . env/bin/activate
```

(Note the space between the dot and the script name. The dot means that this script should run in the context of the current shell. If this command does not work in your shell, try replacing the dot with *source*)

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.

2.2 System Wide Installation

This is possible as well, but I do 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 *sudo*).

2.3 Living on the Edge

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

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

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

This will pull in the dependencies and activate the git head as the current version inside the virtualenv. Then you just have to `git pull origin` to get the latest version.

To just get the development version without git, do this instead:

```
$ mkdir flask
$ cd flask
$ virtualenv env
$ . env/bin/activate
New python executable in env/bin/python
Installing setuptools.....done.
$ easy_install Flask==dev
...
Finished processing dependencies for Flask==dev
```

2.4 *easy_install* on Windows

On Windows, installation of *easy_install* is a little bit trickier because slightly different rules apply on Windows than on Unix-like systems, but it's not difficult. The easiest way to do it is to download the [ez_setup.py](#) file and run it. The easiest way to run the file is to open your downloads folder and double-click on the file.

Next, add the *easy_install* command and other Python scripts to the command search path, by adding your Python installation's Scripts folder to the *PATH* environment variable. To do that, right-click on the "Computer" icon on the Desktop or in the Start menu, and choose "Properties". Then, on Windows Vista and Windows 7 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 Scripts folder; 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 that it worked, open the Command Prompt and execute `easy_install`. If you have User Account Control enabled on Windows Vista or Windows 7, it should prompt you for admin privileges.

QUICKSTART

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

3.1 A Minimal Application

A minimal Flask application looks something like this:

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

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

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://127.0.0.1:5000/
```

Head over to <http://127.0.0.1:5000/>, 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. The first argument is the name of the application's module. If you are using a single module (like here) you should use `__name__` because depending on if it's started as application or imported as module the name will be different (`'__main__'` versus the actual import name). For more information on that, have a look at the Flask documentation.

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.

Externally Visible Server

If you run the server you will notice that the server is only available from your own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary Python code on your computer. If you have *debug* disabled or trust the users on your network, you can make the server publicly available.

Just change the call of the `run()` method to look like this:

```
app.run(host='0.0.0.0')
```

This tells your operating system to listen on a public IP.

3.2 Debug Mode

The `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 application object:

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

Or pass it to `run()`:

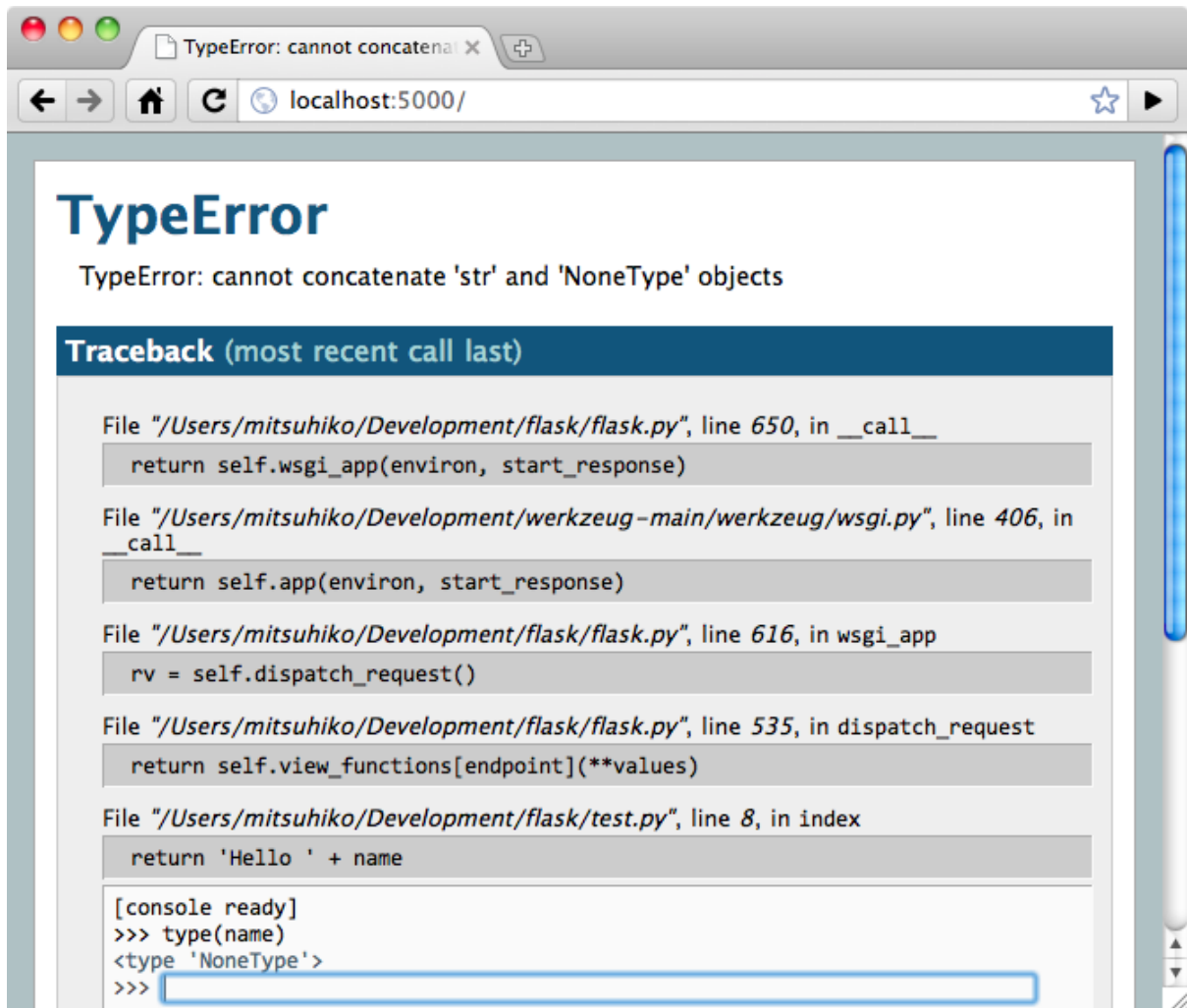
```
app.run(debug=True)
```

Both will have exactly the same effect.

Attention

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

Screenshot of the debugger in action:



Working With Other Debuggers

Debuggers interfere with each other. If you are using another debugger (e.g. PyDev or IntelliJ), you may need to set `app.debug = False`.

3.3 Routing

Modern web applications have beautiful URLs. This helps people remember the URLs which is especially handy for applications that are used from mobile devices with slower network connections. If the user can directly go to the desired page without

having to hit the index page it is more likely they will like the page and come back next time.

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

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

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

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

3.3.1 Variable Rules

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

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

<i>int</i>	accepts integers
<i>float</i>	like <i>int</i> but for floating point values
<i>path</i>	like the default but also accepts slashes

Unique URLs / Redirection Behaviour

Flask's URL rules are based on Werkzeug's routing module. The idea behind that module is to ensure nice looking and also unique URLs based on behaviour Apache and earlier servers coined.

Take these two rules:

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

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

They look rather similar, the difference is the trailing slash in the URL *definition*. In the first case, the canonical URL for the *projects* endpoint has a trailing slash. It's similar to a folder in that sense. Accessing it without a trailing slash will cause Flask to redirect to the canonical URL with the trailing slash.

However in the second case the URL is defined without a slash so it behaves similar to a file and accessing the URL with a trailing slash will be a 404 error.

Why is this? This allows relative URLs to continue working if users access the page when they forget a trailing slash. This behaviour is also consistent with how Apache and other servers work. Also, the URLs will stay unique which helps search engines not indexing the same page twice.

3.3.2 URL Building

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

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

(This also uses the `test_request_context()` method explained below. It basically tells Flask to think we are handling a request even though we are not, we are in an interac-

tive 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.

3.3.3 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 are 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. Likewise as of Flask 0.6, *OPTIONS* is implemented for you as well automatically.

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

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

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

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

POST The browser tells the server that it wants to *post* some new information to that URL and that the server must ensure the data is stored and only stored once. This is how HTML forms 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 is this useful, but there are some good reasons to do it this way. Consider that the connection gets lost during transmission: in this situation a system between the browser and the server might receive the request safely a second time without breaking things. With *POST* that would not be possible because it must only be triggered once.

DELETE Remove the information at the given location.

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

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

3.4 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*.

3.5 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's a simple example of how to render a template:

```
from flask import render_template

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

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

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

Case 1: a module:

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

Case 2: a package:

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

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

Here is an example template:

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

Inside templates you also have access to the request, session and g¹ 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 is a basic introduction to how the Markup class works:

```
>>> from flask import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
```

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

```
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 \xbbb HTML'
```

Changed in version 0.5.

3.6 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:

3.6.1 Context Locals

Insider Information

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

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

Imagine the context being the handling thread. A request comes in and the 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 doing something like unittesting. 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 is an example:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
```

```
# end of the with block, such as basic assertions:
assert request.path == '/hello'
assert request.method == 'POST'
```

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

```
from flask import request

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

3.6.2 The Request Object

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

```
from flask import request
```

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

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    # 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 not user friendly.

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

3.6.3 File Uploads

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

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

```
from flask import request

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

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

```
from flask import request
from werkzeug import secure_filename

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

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

3.6.4 Cookies

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

Reading cookies:

```
from flask import request

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

Storing cookies:

```
from flask import make_response

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

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

For this also see *About Responses*.

3.7 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 use 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 they cannot access (401 means access denied) but it shows how that works.

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

```
from flask import render_template

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

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

3.8 About Responses

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

1. If a response object of the correct type is returned it's directly returned from the view.
2. If it's a string, a response object is created with that data and the default parameters.
3. If a tuple is returned the response object is created by passing the tuple as arguments to the response object's constructor.
4. If neither of that works, Flask will assume the return value is a valid WSGI application and converts that into a response object.

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

Imagine you have a view like this:

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

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

```
@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
    resp.headers['X-Something'] = 'A value'
    return resp
```

3.9 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 signs the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless they know the secret key used for signing.

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

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

```
app = Flask(__name__)
```

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

```
    if 'username' in session:
        return 'Logged in as %s' % escape(session['username'])
    return 'You are not logged in'
```

```
@app.route('/login', methods=['GET', 'POST'])
```

```
def login():
```

```
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form 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)
    return redirect(url_for('index'))
```

```
# set the secret key. keep this really secret:
```

```
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
```

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

How to generate good secret keys

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

```
>>> import os
>>> os.urandom(24)
'\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x010<!\xd5\xa2\xa0\x9fR"\xa1\xa8'
```

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

3.10 Message Flashing

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

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.

3.11 Logging

New in version 0.3. Sometimes you might be in a situation where you deal with data that should be correct, but actually is not. For example you may have some client side code that sends an HTTP request to the server but it's obviously malformed. This might be caused by a user tempering with the data, or the client code failing. Most of the time, it's okay to reply with 400 Bad Request in that situation, but sometimes that won't do and the code has to continue working.

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

Here are some example log calls:

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

The attached logger is a standard logging [Logger](#), so head over to the official [logging documentation](#) for more information.

3.12 Hooking in WSGI Middlewares

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

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

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](#).

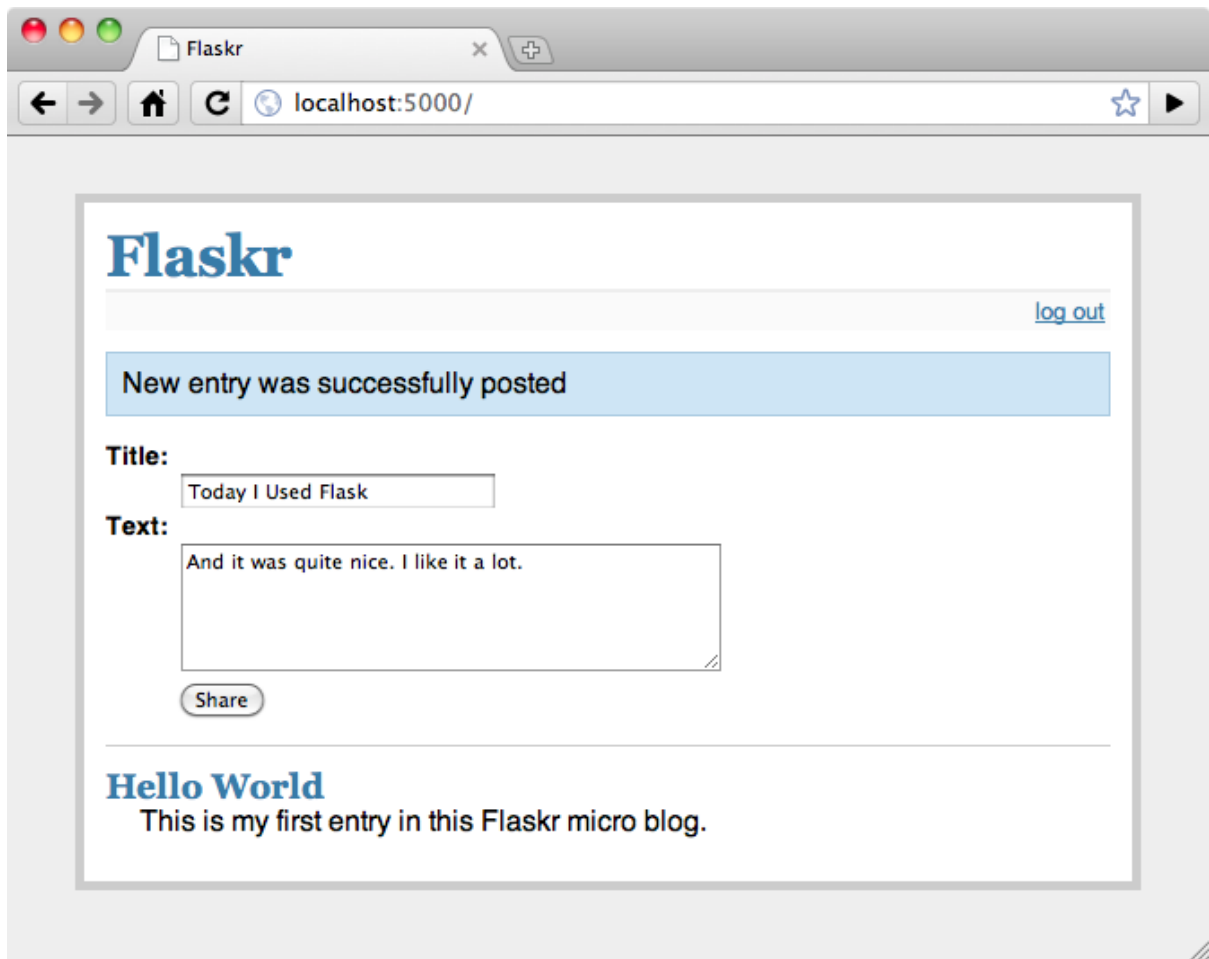
4.1 Introducing Flaskr

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 they can add new entries to the page consisting of a text-only title and some HTML for the text. This HTML is not sanitized because we trust the user here.
3. the 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:



Continue with *Step 0: Creating The Folders*.

4.2 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. The templates you create later in the tutorial will go in this directory.

Continue with *Step 1: Database Schema*.

4.3 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.

Continue with *Step 2: Application Setup Code*.

4.4 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 from the same file:

```
# create our little application :)
app = Flask(__name__)
app.config.from_object(__name__)
```

`from_object()` will look at the given object (if it's a string it will import it) and then look for all uppercase variables defined there. In our case, the configuration we just

wrote a few lines of code above. You can also move that into a separate file.

It is also a good idea to be able to load a configuration from a configurable file. This is what `from_envvar()` can do:

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

That way someone can set an environment variable called `FLASKR_SETTINGS` to specify a config file to be loaded which will then override the default values. The `silent` switch just tells Flask to not complain if no such environment key is set.

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 execute 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(app.config['DATABASE'])
```

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

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

With that out of the way you should be able to start up the application without problems. Do this with the following command:

```
python flaskr.py
```

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

When you head over to the server in your browser you will get 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.

Externally Visible Server

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

Continue with *Step 3: Creating The Database*.

4.5 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()
```

Troubleshooting

If you get an exception later that a table cannot be found check that you did call the `init_db` function and that your table names are correct (singular vs. plural for example).

Continue with *Step 4: Request Database Connections*

4.6 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()`, `after_request()` and `teardown_request()` decorators:

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

@app.teardown_request
def teardown_request(exception):
    g.db.close()
```

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. They are however not guaranteed to be executed if an exception is raised, this is where functions marked with `teardown_request()` come in. They get called after the response has been constructed. They are not allowed to modify the request, and their return values are ignored. If an exception occurred while the request was being processed, it is passed to each function; otherwise, `None` is passed in.

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.

Continue to *Step 5: The View Functions*.

Hint: Where do I put this code?

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

If you need a moment to find your bearings, take a look at how the [example source](#)

is organized. In Flask, you can put all of your application code into a single Python module. You don't have to, and if your app *grows larger*, it's a good idea not to.

4.7 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:

4.7.1 Show Entries

This view shows all the entries stored in the database. It listens on the root of the application and will select title and text from the database. The one with the highest id (the newest entry) will be on top. The rows returned from the cursor 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 in 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)
```

4.7.2 Add New Entry

This view lets the user add new entries if they are logged in. This only responds to *POST* requests, the actual form is shown on the *show_entries* page. If everything worked out well 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*).

Security Note

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

4.7.3 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 addition, a message is flashed that informs the user that he or she was logged in successfully. If an error occurred, the template is notified about that, and the user is asked again:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            error = 'Invalid username'
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_entries'))
    return render_template('login.html', error=error)
```

The logout function, on the other hand, removes that key from the session again. 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 is not in there. This is helpful because now we don't have to check if the user was logged in.

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

Continue with *Step 6: The Templates*.

4.8 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:

4.8.1 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>
```

4.8.2 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 %}

```

4.8.3 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 %}

```

Continue with *Step 7: Adding Style*.

4.9 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; }

```

Continue with *Bonus: Testing the Application*.

4.10 Bonus: Testing the Application

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

TEMPLATES

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

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

5.1 Jinja Setup

Unless customized, Jinja2 is configured by Flask as follows:

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

5.2 Standard Context

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

`config`

The current configuration object (`flask.config`) New in version 0.6.

`request`

The current request object (`flask.request`)

`session`

The current session object (`flask.session`)

`g`

The request-bound object for global variables (`flask.g`)

`url_for()`

The `flask.url_for()` function.

`get_flashed_messages()`

The `flask.get_flashed_messages()` function.

The Jinja Context Behaviour

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

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

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

Importing with context looks like this:

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

5.3 Standard Filters

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

`tojson()`

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

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

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

That the `|tojson` filter escapes forward slashes properly for you.

5.4 Controlling Autoescaping

Autoescaping is the concept of automatically escaping special characters of you. Special characters in the sense of HTML (or XML, and thus XHTML) are `&`, `>`, `<`, `"` as well as `'`. Because these characters carry specific meanings in documents on their own you have to replace them by so called “entities” if you want to use them for text. Not doing

so would not only cause user frustration by the inability to use these characters in text, but can also lead to security problems. (see *Cross-Site Scripting (XSS)*)

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

There are three ways to accomplish that:

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

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

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

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

5.5 Registering Filters

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

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

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

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

In case of the decorator the argument is optional if you want to use the function name as name of the filter.

5.6 Context Processors

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

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

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

TESTING FLASK APPLICATIONS

Something that is untested is broken.

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

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

6.1 The Application

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

6.2 The Testing Skeleton

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

```
import os
import flaskr
import unittest
import tempfile

class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        flaskr.app.config['TESTING'] = True
        self.app = flaskr.app.test_client()
```

```

        flaskr.init_db()

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

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

```

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

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

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

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

```
$ python flaskr_tests.py
```

```

-----
Ran 0 tests in 0.000s

OK

```

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

6.3 The First Test

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

```

class FlaskrTestCase(unittest.TestCase):

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

```

```
def tearDown(self):
    os.close(self.db_fd)
    os.unlink(flaskr.DATABASE)

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

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

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

Run it again and you should see one passing test:

```
$ python flaskr_tests.py
```

```
.
```

```
-----
Ran 1 test in 0.034s
```

```
OK
```

6.4 Logging In and Out

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

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

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

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

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

```
def test_login_logout(self):
    rv = self.login('admin', 'default')
    assert 'You were logged in' in rv.data
```

```

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

```

6.5 Test Adding Messages

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

```

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

```

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

Running that should now give us three passing tests:

```
$ python flaskr_tests.py
```

```
...
```

```
-----
Ran 3 tests in 0.332s
```

```
OK
```

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

6.6 Other Testing Tricks

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

```

app = flask.Flask(__name__)

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

```

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

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

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

```
app = flask.Flask(__name__)

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

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

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

```
app = flask.Flask(__name__)

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

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

6.7 Keeping the Context Around

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

```
app = flask.Flask(__name__)

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

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

functions are already called at this point so your database connection and everything involved is probably already closed down.

HANDLING APPLICATION ERRORS

New in version 0.3. Applications fail, servers fail. Sooner or later you will see an exception in production. Even if your code is 100% correct, you will still see exceptions from time to time. Why? Because everything else involved will fail. Here some situations where perfectly fine code can lead to server errors:

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

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

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

7.1 Error Mails

If the application runs in production mode (which it will do on your server) you won't see any log messages by default. Why is that? Flask tries to be a zero-configuration framework. Where should it drop the logs for you if there is no configuration? Guessing is not a good idea because chances are, the place it guessed is not the place where the user has permission to create a logfile. Also, for most small applications nobody will look at the logs anyways.

In fact, I promise you right now that if you configure a logfile for the application errors you will never look at it except for debugging an issue when a user reported it for you. What you want instead is a mail the second the exception happened. Then you get an alert and you can do something about it.

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

```
ADMINS = ['yourname@example.com']
if not app.debug:
    import logging
    from logging.handlers import SMTPHandler
    mail_handler = SMTPHandler('127.0.0.1',
                               'server-error@example.com',
                               ADMINS, 'YourApplication Failed')
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
```

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

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

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

7.2 Logging to a File

Even if you get mails, you probably also want to log warnings. It’s a good idea to keep as much information around that might be required to debug a problem. Please note that Flask itself will not issue any warnings in the core system, so it’s your responsibility to warn in the code if something seems odd.

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

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

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

```

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

```

7.3 Controlling the Log Format

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

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

Here some example setups:

7.3.1 Email

```

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

Message:

%(message)s
'''))

```

7.3.2 File logging

```

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

```

7.3.3 Complex Log Formatting

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

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

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

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

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

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

For more information, head over to the official documentation.

7.4 Other Libraries

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

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

```
from logging import getLogger
loggers = [app.logger, getLogger('sqlalchemy'),
```

```
        getLogger('otherlibrary')]  
for logger in loggers:  
    logger.addHandler(mail_handler)  
    logger.addHandler(file_handler)
```


CONFIGURATION HANDLING

New in version 0.3. Applications need some kind of configuration. There are different things you might want to change like toggling debug mode, the secret key, and a lot of very similar things.

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

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

8.1 Configuration Basics

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

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

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

```
app.debug = True
```

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

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

8.2 Builtin Configuration Values

The following configuration values are used internally by Flask:

DEBUG	enable/disable debug mode
TESTING	enable/disable testing mode
PROPAGATE_EXCEPTIONS	explicitly enable or disable the propagation of exceptions. If not set or explicitly set to <i>None</i> this is implicitly true if either <i>TESTING</i> or <i>DEBUG</i> is true.
PRESERVE_CONTEXT_ON_EXCEPTION	By default if the application is in debug mode the request context is not popped on exceptions to enable debuggers to introspect the data. This can be disabled by this key. You can also use this setting to force-enable it for non debug execution which might be useful to debug production applications (but also very risky).
SECRET_KEY	the secret key
SESSION_COOKIE_NAME	the name of the session cookie
PERMANENT_SESSION_LIFETIME	the lifetime of a permanent session as <code>datetime.timedelta</code> object.
USE_X_SENDFILE	enable/disable x-sendfile
LOGGER_NAME	the name of the logger
SERVER_NAME	the name and port number of the server. Required for subdomain support (e.g.: 'localhost:5000')
MAX_CONTENT_LENGTH	If set to a value in bytes, Flask will reject incoming requests with a content length greater than this by returning a 413 status code.
TRAP_HTTP_EXCEPTIONS	If this is set to True Flask will not execute the error handlers of HTTP exceptions but instead treat the exception like any other and bubble it through the exception stack. This is helpful for hairy debugging situations where you have to find out where an HTTP exception is coming from.
TRAP_BAD_REQUEST_ERRORS	Werkzeug's internal data structures that deal with request specific data will raise special key errors that are also bad request exceptions. Likewise many operations can implicitly fail with a BadRequest exception for consistency. Since it's nice for debugging to know why exactly it failed this flag can be used to debug those situations. If this config is set to True you will get a regular traceback instead.

More on `SERVER_NAME`

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

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

New in version 0.4: `LOGGER_NAME`New in version 0.5: `SERVER_NAME`New in version 0.6: `MAX_CONTENT_LENGTH`New in version 0.7: `PROPAGATE_EXCEPTIONS`, `PRESERVE_CONTEXT_ON_EXCEPTION`New in version 0.8: `TRAP_BAD_REQUEST_ERRORS`, `TRAP_HTTP_EXCEPTIONS`

8.3 Configuring from Files

Configuration becomes more useful if you can configure from a file, and ideally that file would be outside of the actual application package so that you can install the package with distribute (*Deploying with Distribute*) and still modify that file afterwards.

So a common pattern is this:

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

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

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

On Windows systems use the `set` builtin instead:

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

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

Here is an example configuration file:

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

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

8.4 Configuration Best Practices

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

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

8.5 Development / Production

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

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

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

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

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

```

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

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

class DevelopmentConfig(Config):
    DEBUG = True

class TestingConfig(Config):
    TESTING = True

```

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

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

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

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

SIGNALS

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

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

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

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

9.1 Subscribing to Signals

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

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

Here for example a helper context manager that can be used to figure out in a unittest which templates were rendered and what variables were passed to the template:

```

from flask import template_rendered
from contextlib import contextmanager

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

```

This can now easily be paired with a test client:

```

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

```

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

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

```

from flask import template_rendered

def captured_templates(app, recorded):
    def record(sender, template, context):
        recorded.append((template, context))
    return template_rendered.connected_to(record, app)

```

The example above would then look like this:

```

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

```

Blinker API Changes

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

9.2 Creating Signals

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

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

Now you can create new signals like this:

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

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

For Extension Developers

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

9.3 Sending Signals

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

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

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

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

Passing Proxies as Senders

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

9.4 Decorator Based Signal Subscriptions

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

```
from flask import template_rendered

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

9.5 Core Signals

The following signals exist in Flask:

`flask.template_rendered`

This signal is sent when a template was successfully rendered. The signal is invoked with the instance of the template as *template* and the context as dictionary (named *context*).

Example subscriber:

```
def log_template_renders(sender, template, context):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
                        context)
```

```
from flask import template_rendered
template_rendered.connect(log_template_renders, app)
```

`flask.request_started`

This signal is sent before any request processing started but when the request context was set up. Because the request context is already bound, the subscriber can access the request with the standard global proxies such as `request`.

Example subscriber:

```
def log_request(sender):
    sender.logger.debug('Request context is set up')
```

```
from flask import request_started
request_started.connect(log_request, app)
```

`flask.request_finished`

This signal is sent right before the response is sent to the client. It is passed the response to be sent named *response*.

Example subscriber:

```
def log_response(sender, response):
    sender.logger.debug('Request context is about to close down.  '
                       'Response: %s', response)
```

```
from flask import request_finished
request_finished.connect(log_response, app)
```

flask.got_request_exception

This signal is sent when an exception happens during request processing. It is sent *before* the standard exception handling kicks in and even in debug mode, where no exception handling happens. The exception itself is passed to the subscriber as *exception*.

Example subscriber:

```
def log_exception(sender, exception):
    sender.logger.debug('Got exception during processing: %s', exception)
```

```
from flask import got_request_exception
got_request_exception.connect(log_exception, app)
```

flask.request_tearing_down

This signal is sent when the request is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

```
def close_db_connection(sender):
    session.close()
```

```
from flask import request_tearing_down
request_tearing_down.connect(close_db_connection, app)
```


PLUGGABLE VIEWS

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

10.1 Basic Principle

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

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

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

```
from flask.views import View

class ShowUsers(View):

    def dispatch_request(self):
        users = User.query.all()
        return render_template('users.html', objects=users)

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

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

```

from flask.views import View

class ListView(View):

    def get_template_name(self):
        raise NotImplementedError()

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

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

class UserView(ListView):

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

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

```

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

```

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

```

And then you can register it like this:

```

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

```

10.2 Method Hints

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

```

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

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

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

```

10.3 Method Based Dispatching

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

```

from flask.views import MethodView

class UserAPI(MethodView):

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

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

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

```

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

THE REQUEST CONTEXT

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

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

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

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

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

The magic that makes this works is internally referred in Flask as the “request context”.

11.1 Diving into Context Locals

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

```
from flask import request, url_for

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

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

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

That makes a lot of sense because we currently do not have a request we could access. So we have to make a request and bind it to the current context. The `test_request_context` method can create us a `RequestContext`:

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

```

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

```
>>> ctx.push()

```

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

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

```

Until you call *pop*:

```
>>> ctx.pop()

```

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

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

11.2 How the Context Works

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

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

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

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

11.3 Callbacks and Errors

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

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

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

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

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

11.4 Teardown Callbacks

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

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

```

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

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

```

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

```

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

```

11.5 Notes On Proxies

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

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

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

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

```

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

```

11.6 Context Preservation on Error

If an error occurs or not, at the end of the request the request context is popped and all data associated with it is destroyed. During development however that can be

problematic as you might want to have the information around for a longer time in case an exception occurred. In Flask 0.6 and earlier in debug mode, if an exception occurred, the request context was not popped so that the interactive debugger can still provide you with important information.

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

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

MODULAR APPLICATIONS WITH BLUEPRINTS

New in version 0.7. Flask uses a concept of *blueprints* for making application components and supporting common patterns within an application or across applications. Blueprints can greatly simplify how large applications work and provide a central means for Flask extensions to register operations on applications. A Blueprint object works similarly to a Flask application object, but it is not actually an application. Rather it is a *blueprint* of how to construct or extend an application.

12.1 Why Blueprints?

Blueprints in Flask are intended for these cases:

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

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

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

12.2 The Concept of Blueprints

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

12.3 My First Blueprint

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

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

simple_page = Blueprint('simple_page', __name__)

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

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

12.4 Registering Blueprints

So how do you register that blueprint? Like this:

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

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

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

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

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

Blueprints however can also be mounted at different locations:

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

And sure enough, these are the generated rules:

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

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

12.5 Blueprint Resources

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

12.5.1 Blueprint Resource Folder

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

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

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

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

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

12.5.2 Static Files

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

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

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

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

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

12.5.3 Templates

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

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

As for static files, the path can be absolute or relative to the blueprint resource folder. The template folder is added to the searchpath of templates but with a lower priority than the actual application's template folder. That way you can easily override templates that a blueprint provides in the actual application.

So if you have a blueprint in the folder *yourapplication/admin* and you want to render the template *'admin/index.html'* and you have provided templates as a *template_folder* you will have to create a file like this: *yourapplication/admin/templates/admin/index.html*.

12.6 Building URLs

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

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

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

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

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

WORKING WITH THE SHELL

New in version 0.3. One of the reasons everybody loves Python is the interactive shell. It basically allows you to execute Python commands in real time and immediately get results back. Flask itself does not come with an interactive shell, because it does not require any specific setup upfront, just import your application and start playing around.

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

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

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

13.1 Creating a Request Context

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

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

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

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

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

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

13.2 Firing Before/After Request

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

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

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

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

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

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

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

13.3 Further Improving the Shell Experience

If you like the idea of experimenting in a shell, create yourself a module with stuff you want to star import into your interactive session. There you could also define some more helper methods for common things such as initializing the database, dropping tables etc.

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

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

PATTERNS FOR FLASK

Certain things are common enough that the chances are high you will find them in most web applications. For example quite a lot of applications are using relational databases and user authentication. In that case, chances are they will open a database connection at the beginning of the request and get the information of the currently logged in user. At the end of the request, the database connection is closed again.

There are more user contributed snippets and patterns in the [Flask Snippet Archives](#).

14.1 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
    ...
```

14.1.1 Simple Packages

To convert that into a larger one, just create a new folder *yourapplication* inside the existing one and move everything below it. Then rename *yourapplication.py* to *__init__.py*. (Make sure to delete all *.pyc* files first, otherwise things would most likely break)

You should then end up with something like that:

```
/yourapplication
  /yourapplication
    /__init__.py
    /static
```

```
    /style.css
/templates
    layout.html
    index.html
    login.html
    ...
```

But how do you run your application now? The naive python `yourapplication/__init__.py` will not work. Let's just say that Python does not want modules in packages to be the startup file. But that is not a big problem, just add a new file called *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 has to be in the `__init__.py` file. That way each module can import it safely and the `__name__` variable will resolve to the correct package.
2. all the view functions (the ones with a `route()` decorator on top) have to be imported when in the `__init__.py` file. Not the object itself, but the module it is in. Import the view module **after the application object is created**.

Here's an example `__init__.py`:

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

import yourapplication.views
```

And this is what *views.py* would look like:

```
from yourapplication import app

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

You should then end up with something like that:

```
/yourapplication
/runserver.py
/yourapplication
    /__init__.py
    /views.py
    /static
        /style.css
    /templates
        layout.html
```

```
index.html
login.html
...
```

Circular Imports

Every Python programmer hates them, and yet we just added some: circular imports (That's when two modules depend on each other. In this case *views.py* depends on *__init__.py*). Be advised that this is a bad idea in general but here it is actually fine. The reason for this is that we are not actually using the views in *__init__.py* and just ensuring the module is imported and we are doing that at the bottom of the file.

There are still some problems with that approach but if you want to use decorators there is no way around that. Check out the *Becoming Big* section for some inspiration how to deal with that.

14.1.2 Working with Blueprints

If you have larger applications it's recommended to divide them into smaller groups where each group is implemented with the help of a blueprint. For a gentle introduction into this topic refer to the *Modular Applications with Blueprints* chapter of the documentation.

14.2 Application Factories

If you are already using packages and blueprints for your application (*Modular Applications with Blueprints*) there are a couple of really nice ways to further improve the experience. A common pattern is creating the application object when the blueprint is imported. But if you move the creation of this object, into a function, you can then create multiple instances of this and later.

So why would you want to do this?

1. Testing. You can have instances of the application with different settings to test every case.
2. Multiple instances. Imagine you want to run different versions of the same application. Of course you could have multiple instances with different configs set up in your webserver, but if you use factories, you can have multiple instances of the same application running in the same application process which can be handy.

So how would you then actually implement that?

14.2.1 Basic Factories

The idea is to set up the application in a function. Like this:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.views.admin import admin
    from yourapplication.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)

    return app
```

The downside is that you cannot use the application object in the blueprints at import time. You can however use it from within a request. How do you get access to the application with the config? Use `current_app`:

```
from flask import current_app, Blueprint, render_template
admin = Blueprint('admin', __name__, url_prefix='/admin')

@admin.route('/')
def index():
    return render_template(current_app.config['INDEX_TEMPLATE'])
```

Here we look up the name of a template in the config.

14.2.2 Using Applications

So to use such an application you then have to create the application first. Here an example `run.py` file that runs such an application:

```
from yourapplication import create_app
app = create_app('/path/to/config.cfg')
app.run()
```

14.2.3 Factory Improvements

The factory function from above is not very clever so far, you can improve it. The following changes are straightforward and possible:

1. make it possible to pass in configuration values for unittests so that you don't have to create config files on the filesystem
2. call a function from a blueprint when the application is setting up so that you have a place to modify attributes of the application (like hooking in before / after request handlers etc.)
3. Add in WSGI middlewares when the application is creating if necessary.

14.3 Application Dispatching

Application dispatching is the process of combining multiple Flask applications on the WSGI level. You can not only combine Flask applications into something larger but any WSGI application. This would even allow you to run a Django and a Flask application in the same interpreter side by side if you want. The usefulness of this depends on how the applications work internally.

The fundamental difference from the *module approach* is that in this case you are running the same or different Flask applications that are entirely isolated from each other. They run different configurations and are dispatched on the WSGI level.

14.3.1 Working with this Document

Each of the techniques and examples below results in an application object that can be run with any WSGI server. For production, see *Deployment Options*. For development, Werkzeug provides a builtin server for development available at `werkzeug.serving.run_simple()`:

```
from werkzeug.serving import run_simple
run_simple('localhost', 5000, application, use_reloader=True)
```

Note that `run_simple` is not intended for use in production. Use a *full-blown WSGI server*.

14.3.2 Combining Applications

If you have entirely separated applications and you want them to work next to each other in the same Python interpreter process you can take advantage of the `werkzeug.wsgi.DispatcherMiddleware`. The idea here is that each Flask application is a valid WSGI application and they are combined by the dispatcher middleware into a larger one that dispatched based on prefix.

For example you could have your main application run on `/` and your backend interface on `/backend`:

```
from werkzeug.wsgi import DispatcherMiddleware
from frontend_app import application as frontend
from backend_app import application as backend

application = DispatcherMiddleware(frontend, {
    '/backend': backend
})
```

14.3.3 Dispatch by Subdomain

Sometimes you might want to use multiple instances of the same application with different configurations. Assuming the application is created inside a function and you can call that function to instantiate it, that is really easy to implement. In order to develop your application to support creating new instances in functions have a look at the *Application Factories* pattern.

A very common example would be creating applications per subdomain. For instance you configure your webserver to dispatch all requests for all subdomains to your application and you then use the subdomain information to create user-specific instances. Once you have your server set up to listen on all subdomains you can use a very simple WSGI application to do the dynamic application creation.

The perfect level for abstraction in that regard is the WSGI layer. You write your own WSGI application that looks at the request that comes and delegates it to your Flask application. If that application does not exist yet, it is dynamically created and remembered:

```
from threading import Lock

class SubdomainDispatcher(object):

    def __init__(self, domain, create_app):
        self.domain = domain
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, host):
        host = host.split(':')[0]
        assert host.endswith(self.domain), 'Configuration error'
        subdomain = host[:-len(self.domain)].rstrip('.')
        with self.lock:
            app = self.instances.get(subdomain)
            if app is None:
                app = self.create_app(subdomain)
                self.instances[subdomain] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(environ['HTTP_HOST'])
        return app(environ, start_response)
```

This dispatcher can then be used like this:

```
from myapplication import create_app, get_user_for_subdomain
from werkzeug.exceptions import NotFound

def make_app(subdomain):
    user = get_user_for_subdomain(subdomain)
    if user is None:
```

```

        # if there is no user for that subdomain we still have
        # to return a WSGI application that handles that request.
        # We can then just return the NotFound() exception as
        # application which will render a default 404 page.
        # You might also redirect the user to the main page then
        return NotFound()

    # otherwise create the application for the specific user
    return create_app(user)

application = SubdomainDispatcher('example.com', make_app)

```

14.3.4 Dispatch by Path

Dispatching by a path on the URL is very similar. Instead of looking at the *Host* header to figure out the subdomain one simply looks at the request path up to the first slash:

```

from threading import Lock
from werkzeug.wsgi import pop_path_info, peek_path_info

class PathDispatcher(object):

    def __init__(self, default_app, create_app):
        self.default_app = default_app
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, prefix):
        with self.lock:
            app = self.instances.get(prefix)
            if app is None:
                app = self.create_app(prefix)
                if app is not None:
                    self.instances[prefix] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(peek_path_info(environ))
        if app is not None:
            pop_path_info(environ)
        else:
            app = self.default_app
        return app(environ, start_response)

```

The big difference between this and the subdomain one is that this one falls back to another application if the creator function returns *None*:

```

from myapplication import create_app, default_app, get_user_for_prefix

def make_app(prefix):
    user = get_user_for_prefix(prefix)
    if user is not None:
        return create_app(user)

application = PathDispatcher(default_app, make_app)

```

14.4 Using URL Processors

New in version 0.7. Flask 0.7 introduces the concept of URL processors. The idea is that you might have a bunch of resources with common parts in the URL that you don't always explicitly want to provide. For instance you might have a bunch of URLs that have the language code in it but you don't want to have to handle it in every single function yourself.

URL processors are especially helpful when combined with blueprints. We will handle both application specific URL processors here as well as blueprint specifics.

14.4.1 Internationalized Application URLs

Consider an application like this:

```

from flask import Flask, g

app = Flask(__name__)

@app.route('/<lang_code>/')
def index(lang_code):
    g.lang_code = lang_code
    ...

@app.route('/<lang_code>/about')
def about(lang_code):
    g.lang_code = lang_code
    ...

```

This is an awful lot of repetition as you have to handle the language code setting on the `g` object yourself in every single function. Sure, a decorator could be used to simplify this, but if you want to generate URLs from one function to another you would have to still provide the language code explicitly which can be annoying.

For the latter, this is where `url_defaults()` functions come in. They can automatically inject values into a call for `url_for()` automatically. The code below checks if the language code is not yet in the dictionary of URL values and if the endpoint wants a value named `'lang_code'`:

```
@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code
```

The method `is_endpoint expecting()` of the URL map can be used to figure out if it would make sense to provide a language code for the given endpoint.

The reverse of that function are `url_value_preprocessor()`s. They are executed right after the request was matched and can execute code based on the URL values. The idea is that they pull information out of the values dictionary and put it somewhere else:

```
@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)
```

That way you no longer have to do the *lang_code* assignment to `g` in every function. You can further improve that by writing your own decorator that prefixes URLs with the language code, but the more beautiful solution is using a blueprint. Once the `'lang_code'` is popped from the values dictionary and it will no longer be forwarded to the view function reducing the code to this:

```
from flask import Flask, g

app = Flask(__name__)

@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code

@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)

@app.route('/<lang_code>/')
def index():
    ...

@app.route('/<lang_code>/about')
def about():
    ...
```

14.4.2 Internationalized Blueprint URLs

Because blueprints can automatically prefix all URLs with a common string it's easy to automatically do that for every function. Furthermore blueprints can have per-blueprint URL processors which removes a whole lot of logic from the `url_defaults()` function because it no longer has to check if the URL is really interested in a `'lang_code'` parameter:

```
from flask import Blueprint, g

bp = Blueprint('frontend', __name__, url_prefix='<lang_code>')

@bp.url_defaults
def add_language_code(endpoint, values):
    values.setdefault('lang_code', g.lang_code)

@bp.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code')

@bp.route('/')
def index():
    ...

@bp.route('/about')
def about():
    ...
```

14.5 Deploying with Distribute

`distribute`, formerly `setuptools`, is an extension library that is commonly used to (like the name says) distribute Python libraries and extensions. It extends `distutils`, a basic module installation system shipped with Python to also support various more complex constructs that make larger applications easier to distribute:

- **support for dependencies:** a library or application can declare a list of other libraries it depends on which will be installed automatically for you.
- **package registry:** `setuptools` registers your package with your Python installation. This makes it possible to query information provided by one package from another package. The best known feature of this system is the entry point support which allows one package to declare an “entry point” another package can hook into to extend the other package.
- **installation manager:** `easy_install`, which comes with `distribute` can install other libraries for you. You can also use `pip` which sooner or later will replace `easy_install` which does more than just installing packages for you.

Flask itself, and all the libraries you can find on the cheeseshop are distributed with either `distribute`, the older `setuptools` or `distutils`.

In this case we assume your application is called *yourapplication.py* and you are not using a module, but a *package*. Distributing resources with standard modules is not supported by *distribute* so we will not bother with it. If you have not yet converted your application into a package, head over to the *Larger Applications* pattern to see how this can be done.

A working deployment with *distribute* is the first step into more complex and more automated deployment scenarios. If you want to fully automate the process, also read the *Deploying with Fabric* chapter.

14.5.1 Basic Setup Script

Because you have Flask running, you either have *setuptools* or *distribute* available on your system anyways. If you do not, fear not, there is a script to install it for you: *distribute_setup.py*. Just download and run with your Python interpreter.

Standard disclaimer applies: *you better use a virtualenv*.

Your setup code always goes into a file named *setup.py* next to your application. The name of the file is only convention, but because everybody will look for a file with that name, you better not change it.

Yes, even if you are using *distribute*, you are importing from a package called *setuptools*. *distribute* is fully backwards compatible with *setuptools*, so it also uses the same import name.

A basic *setup.py* file for a Flask application looks like this:

```
from setuptools import setup

setup(
    name='Your Application',
    version='1.0',
    long_description=__doc__,
    packages=['yourapplication'],
    include_package_data=True,
    zip_safe=False,
    install_requires=['Flask']
)
```

Please keep in mind that you have to list subpackages explicitly. If you want *distribute* to lookup the packages for you automatically, you can use the *find_packages* function:

```
from setuptools import setup, find_packages

setup(
    ...
    packages=find_packages()
)
```

Most parameters to the *setup* function should be self explanatory, *include_package_data* and *zip_safe* might not be. *include_package_data* tells *distribute* to look for a MANI-

MANIFEST.in file and install all the entries that match as package data. We will use this to distribute the static files and templates along with the Python module (see *Distributing Resources*). The *zip_safe* flag can be used to force or prevent zip Archive creation. In general you probably don't want your packages to be installed as zip files because some tools do not support them and they make debugging a lot harder.

14.5.2 Distributing Resources

If you try to install the package you just created, you will notice that folders like *static* or *templates* are not installed for you. The reason for this is that distribute does not know which files to add for you. What you should do, is to create a *MANIFEST.in* file next to your *setup.py* file. This file lists all the files that should be added to your tarball:

```
recursive-include yourapplication/templates *
recursive-include yourapplication/static *
```

Don't forget that even if you enlist them in your *MANIFEST.in* file, they won't be installed for you unless you set the *include_package_data* parameter of the *setup* function to *True*!

14.5.3 Declaring Dependencies

Dependencies are declared in the *install_requires* parameter as list. Each item in that list is the name of a package that should be pulled from PyPI on installation. By default it will always use the most recent version, but you can also provide minimum and maximum version requirements. Here some examples:

```
install_requires=[
    'Flask>=0.2',
    'SQLAlchemy>=0.6',
    'BrokenPackage>=0.7,<=1.0'
]
```

I mentioned earlier that dependencies are pulled from PyPI. What if you want to depend on a package that cannot be found on PyPI and won't be because it is an internal package you don't want to share with anyone? Just still do as if there was a PyPI entry for it and provide a list of alternative locations where distribute should look for tarballs:

```
dependency_links=['http://example.com/yourfiles']
```

Make sure that page has a directory listing and the links on the page are pointing to the actual tarballs with their correct filenames as this is how distribute will find the files. If you have an internal company server that contains the packages, provide the URL to that server there.

14.5.4 Installing / Developing

To install your application (ideally into a virtualenv) just run the *setup.py* script with the *install* parameter. It will install your application into the virtualenv's site-packages folder and also download and install all dependencies:

```
$ python setup.py install
```

If you are developing on the package and also want the requirements to be installed, you can use the *develop* command instead:

```
$ python setup.py develop
```

This has the advantage of just installing a link to the site-packages folder instead of copying the data over. You can then continue to work on the code without having to run *install* again after each change.

14.6 Deploying with Fabric

Fabric is a tool for Python similar to Makefiles but with the ability to execute commands on a remote server. In combination with a properly set up Python package (*Larger Applications*) and a good concept for configurations (*Configuration Handling*) it is very easy to deploy Flask applications to external servers.

Before we get started, here a quick checklist of things we have to ensure upfront:

- Fabric 1.0 has to be installed locally. This tutorial assumes the latest version of Fabric.
- The application already has to be a package and requires a working *setup.py* file (*Deploying with Distribute*).
- In the following example we are using *mod_wsgi* for the remote servers. You can of course use your own favourite server there, but for this example we chose Apache + *mod_wsgi* because it's very easy to setup and has a simple way to reload applications without root access.

14.6.1 Creating the first Fabfile

A fabfile is what controls what Fabric executes. It is named *fabfile.py* and executed by the *fab* command. All the functions defined in that file will show up as *fab* subcommands. They are executed on one or more hosts. These hosts can be defined either in the fabfile or on the command line. In this case we will add them to the fabfile.

This is a basic first example that has the ability to upload the current sourcecode to the server and install it into a pre-existing virtual environment:

```
from fabric.api import *
```

```

# the user to use for the remote commands
env.user = 'appuser'
# the servers where the commands are executed
env.hosts = ['server1.example.com', 'server2.example.com']

def pack():
    # create a new source distribution as tarball
    local('python setup.py sdist --formats=gztar', capture=False)

def deploy():
    # figure out the release name and version
    dist = local('python setup.py --fullname', capture=True).strip()
    # upload the source tarball to the temporary folder on the server
    put('dist/%s.tar.gz' % dist, '/tmp/yourapplication.tar.gz')
    # create a place where we can unzip the tarball, then enter
    # that directory and unzip it
    run('mkdir /tmp/yourapplication')
    with cd('/tmp/yourapplication'):
        run('tar xzf /tmp/yourapplication.tar.gz')
        # now setup the package with our virtual environment's
        # python interpreter
        run('/var/www/yourapplication/env/bin/python setup.py install')
    # now that all is set up, delete the folder again
    run('rm -rf /tmp/yourapplication /tmp/yourapplication.tar.gz')
    # and finally touch the .wsgi file so that mod_wsgi triggers
    # a reload of the application
    run('touch /var/www/yourapplication.wsgi')

```

The example above is well documented and should be straightforward. Here a recap of the most common commands fabric provides:

- *run* - executes a command on a remote server
- *local* - executes a command on the local machine
- *put* - uploads a file to the remote server
- *cd* - changes the directory on the serverside. This has to be used in combination with the *with* statement.

14.6.2 Running Fabfiles

Now how do you execute that fabfile? You use the *fab* command. To deploy the current version of the code on the remote server you would use this command:

```
$ fab pack deploy
```

However this requires that our server already has the `/var/www/yourapplication` folder created and `/var/www/yourapplication/env` to be a virtual environment. Furthermore are we not creating the configuration or `.wsgi` file on the server. So how do we bootstrap a new server into our infrastructure?

This now depends on the number of servers we want to set up. If we just have one application server (which the majority of applications will have), creating a command in the fabfile for this is overkill. But obviously you can do that. In that case you would probably call it *setup* or *bootstrap* and then pass the servername explicitly on the command line:

```
$ fab -H newserver.example.com bootstrap
```

To setup a new server you would roughly do these steps:

1. Create the directory structure in `/var/www`:

```
$ mkdir /var/www/yourapplication
$ cd /var/www/yourapplication
$ virtualenv --distribute env
```

2. Upload a new *application.wsgi* file to the server and the configuration file for the application (eg: *application.cfg*)
3. Create a new Apache config for *yourapplication* and activate it. Make sure to activate watching for changes of the *.wsgi* file so that we can automatically reload the application by touching it. (See *mod_wsgi (Apache)* for more information)

So now the question is, where do the *application.wsgi* and *application.cfg* files come from?

14.6.3 The WSGI File

The WSGI file has to import the application and also to set an environment variable so that the application knows where to look for the config. This is a short example that does exactly that:

```
import os
os.environ['YOURAPPLICATION_CONFIG'] = '/var/www/yourapplication/application.cfg'
from yourapplication import app
```

The application itself then has to initialize itself like this to look for the config at that environment variable:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_config')
app.config.from_envvar('YOURAPPLICATION_CONFIG')
```

This approach is explained in detail in the *Configuration Handling* section of the documentation.

14.6.4 The Configuration File

Now as mentioned above, the application will find the correct configuration file by looking up the *YOURAPPLICATION_CONFIG* environment variable. So we have to

put the configuration in a place where the application will be able to find it. Configuration files have the unfriendly quality of being different on all computers, so you do not version them usually.

A popular approach is to store configuration files for different servers in a separate version control repository and check them out on all servers. Then symlink the file that is active for the server into the location where it's expected (eg: `/var/www/yourapplication`).

Either way, in our case here we only expect one or two servers and we can upload them ahead of time by hand.

14.6.5 First Deployment

Now we can do our first deployment. We have set up the servers so that they have their virtual environments and activated apache configs. Now we can pack up the application and deploy it:

```
$ fab pack deploy
```

Fabric will now connect to all servers and run the commands as written down in the `fabfile`. First it will execute `pack` so that we have our tarball ready and then it will execute `deploy` and upload the source code to all servers and install it there. Thanks to the `setup.py` file we will automatically pull in the required libraries into our virtual environment.

14.6.6 Next Steps

From that point onwards there is so much that can be done to make deployment actually fun:

- Create a *bootstrap* command that initializes new servers. It could initialize a new virtual environment, setup apache appropriately etc.
- Put configuration files into a separate version control repository and symlink the active configs into place.
- You could also put your application code into a repository and check out the latest version on the server and then install. That way you can also easily go back to older versions.
- hook in testing functionality so that you can deploy to an external server and run the testsuite.

Working with Fabric is fun and you will notice that it's quite magical to type `fab deploy` and see your application being deployed automatically to one or more remote servers.

14.7 Using SQLite 3 with Flask

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

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

```
import sqlite3
from flask import g

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

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

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

@app.teardown_request
def teardown_request(exception):
    g.db.close()
```

14.7.1 Connect on Demand

The downside of this approach is that this will only work if Flask executed the `before_request` handlers for you. If you are attempting to use the database from a script or the interactive Python shell you would have to do something like this:

```
with app.test_request_context():
    app.preprocess_request()
    # now you can use the g.db object
```

In order to trigger the execution of the connection code. You won't be able to drop the dependency on the request context this way, but you could make it so that the application connects when necessary:

```
def get_connection():
    db = getattr(g, '_db', None)
    if db is None:
        db = g._db = connect_db()
    return db
```

Downside here is that you have to use `db = get_connection()` instead of just being able to use `g.db` directly.

14.7.2 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 formatting because this makes it possible to attack the application using [SQL Injections](#).

14.7.3 Initial Schemas

Relational databases need schemas, so applications often ship a *schema.sql* file that creates the database. It's a good idea to provide a function that creates the database based on that schema. This function can do that for you:

```
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()
```

14.8 SQLAlchemy in Flask

Many people prefer [SQLAlchemy](#) for database access. In this case it's encouraged to use a package instead of a module for your flask application and drop the models into a separate module (*Larger Applications*). While that is not necessary, it makes a lot of sense.

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

14.8.1 Flask-SQLAlchemy Extension

Because SQLAlchemy is a common database abstraction layer and object relational mapper that requires a little bit of configuration effort, there is a Flask extension that handles that for you. This is recommended if you want to get started quickly.

You can download [Flask-SQLAlchemy](#) from [PyPI](#).

14.8.2 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', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))

Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    # import all modules here that might define models so that
    # they will be registered properly on the metadata. Otherwise
    # you will have to import them first before calling init_db()
    import yourapplication.models
    Base.metadata.create_all(bind=engine)
```

To define your models, just subclass the *Base* class that was created by the code above. If you are wondering why we don't have to care about threads here (like we did in the SQLite3 example above with the *g* object): that's because SQLAlchemy does that for us already with the *scoped_session*.

To use SQLAlchemy in a declarative way with your application, you just have to put the following code into your application module. Flask will automatically remove database sessions at the end of the request for you:

```
from yourapplication.database import db_session
```

```
@app.teardown_request
def shutdown_session(exception=None):
    db_session.remove()
```

Here is an example model (put this into *models.py*, e.g.):

```
from sqlalchemy import Column, Integer, String
from yourapplication.database import Base

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True)
    email = Column(String(120), unique=True)

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

    def __repr__(self):
        return '<User %r>' % (self.name)
```

To create the database you can use the *init_db* function:

```
>>> from yourapplication.database import init_db
>>> init_db()
```

You can insert entries into the database like this:

```
>>> from yourapplication.database import db_session
>>> from yourapplication.models import User
>>> u = User('admin', 'admin@localhost')
>>> db_session.add(u)
>>> db_session.commit()
```

Querying is simple as well:

```
>>> User.query.all()
[<User u'admin'>]
>>> User.query.filter(User.name == 'admin').first()
<User u'admin'>
```

14.8.3 Manual Object Relational Mapping

Manual object relational mapping has a few upsides and a few downsides versus the declarative approach from above. The main difference is that you define tables and

classes separately and map them together. It's more flexible but a little more to type. In general it works like the declarative approach, so make sure to also split up your application into multiple modules in a package.

Here is an example *database.py* module for your application:

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine('sqlite:///tmp/test.db', convert_unicode=True)
metadata = MetaData()
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))

def init_db():
    metadata.create_all(bind=engine)
```

As for the declarative approach you need to close the session after each request. Put this into your application module:

```
from yourapplication.database import db_session

@app.teardown_request
def shutdown_session(exception=None):
    db_session.remove()
```

Here is an example table and model (put this into *models.py*):

```
from sqlalchemy import Table, Column, Integer, String
from sqlalchemy.orm import mapper
from yourapplication.database import metadata, db_session

class User(object):
    query = db_session.query_property()

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

    def __repr__(self):
        return '<User %r>' % (self.name, self.email)

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('name', String(50), unique=True),
              Column('email', String(120), unique=True)
)
mapper(User, users)
```

Querying and inserting works exactly the same as in the example above.

14.8.4 SQL Abstraction Layer

If you just want to use the database system (and SQL) abstraction layer you basically only need the engine:

```
from sqlalchemy import create_engine, MetaData

engine = create_engine('sqlite:///tmp/test.db', convert_unicode=True)
metadata = MetaData(bind=engine)
```

Then you can either declare the tables in your code like in the examples above, or automatically load them:

```
users = Table('users', metadata, autoload=True)
```

To insert data you can use the *insert* method. We have to get a connection first so that we can use a transaction:

```
>>> con = engine.connect()
>>> con.execute(users.insert(name='admin', email='admin@localhost'))
```

SQLAlchemy will automatically commit for us.

To query your database, you use the engine directly or use a connection:

```
>>> users.select(users.c.id == 1).execute().first()
(1, u'admin', u'admin@localhost')
```

These results are also dict-like tuples:

```
>>> r = users.select(users.c.id == 1).execute().first()
>>> r['name']
u'admin'
```

You can also pass strings of SQL statements to the *execute()* method:

```
>>> engine.execute('select * from users where id = :1', [1]).first()
(1, u'admin', u'admin@localhost')
```

For more information about SQLAlchemy, head over to the [website](#).

14.9 Uploading Files

Ah yes, the good old problem of file uploads. The basic idea of file uploads is actually quite simple. It basically works like this:

1. A `<form>` tag is marked with `enctype=multipart/form-data` and an `<input type=file>` is placed in that form.
2. The application accesses the file from the files dictionary on the request object.
3. use the *save()* method of the file to save the file permanently somewhere on the filesystem.

14.9.1 A Gentle Introduction

Let's start with a very basic application that uploads a file to a specific upload folder and displays a file to the user. Let's look at the bootstrapping code for our application:

```
import os
from flask import Flask, request, redirect, url_for
from werkzeug import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])

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

So first we need a couple of imports. Most should be straightforward, the `werkzeug.secure_filename()` is explained a little bit later. The `UPLOAD_FOLDER` is where we will store the uploaded files and the `ALLOWED_EXTENSIONS` is the set of allowed file extensions. Then we add a URL rule by hand to the application. Now usually we're not doing that, so why here? The reason is that we want the webserver (or our development server) to serve these files for us and so we only need a rule to generate the URL to these files.

Why do we limit the extensions that are allowed? You probably don't want your users to be able to upload everything there if the server is directly sending out the data to the client. That way you can make sure that users are not able to upload HTML files that would cause XSS problems (see *Cross-Site Scripting (XSS)*). Also make sure to disallow `.php` files if the server executes them, but who has PHP installed on his server, right? :)

Next the functions that check if an extension is valid and that uploads the file and redirects the user to the URL for the uploaded file:

```
def allowed_file(filename):
    return '.' in filename and \
        filename.rsplit('.', 1)[1] in ALLOWED_EXTENSIONS

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('uploaded_file',
                                    filename=filename))

    return '''
<!doctype html>
<title>Upload new File</title>
<h1>Upload new File</h1>
<form action="" method=post enctype=multipart/form-data>
  <p><input type=file name=file>
    <input type=submit value=Upload>
```

```
</form>
'''
```

So what does that `secure_filename()` function actually do? Now the problem is that there is that principle called “never trust user input”. This is also true for the filename of an uploaded file. All submitted form data can be forged, and filenames can be dangerous. For the moment just remember: always use that function to secure a filename before storing it directly on the filesystem.

Information for the Pros

So you’re interested in what that `secure_filename()` function does and what the problem is if you’re not using it? So just imagine someone would send the following information as *filename* to your application:

```
filename = "../../../home/username/.bashrc"
```

Assuming the number of `../` is correct and you would join this with the `UPLOAD_FOLDER` the user might have the ability to modify a file on the server’s filesystem he or she should not modify. This does require some knowledge about how the application looks like, but trust me, hackers are patient :)

Now let’s look how that function works:

```
>>> secure_filename('../../../home/username/.bashrc')
'home_username_.bashrc'
```

Now one last thing is missing: the serving of the uploaded files. As of Flask 0.5 we can use a function that does that for us:

```
from flask import send_from_directory

@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                              filename)
```

Alternatively you can register *uploaded_file* as *build_only* rule and use the `SharedDataMiddleware`. This also works with older versions of Flask:

```
from werkzeug import SharedDataMiddleware
app.add_url_rule('/uploads/<filename>', 'uploaded_file',
                 build_only=True)
app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/uploads': app.config['UPLOAD_FOLDER']
})
```

If you now run the application everything should work as expected.

14.9.2 Improving Uploads

New in version 0.6. So how exactly does Flask handle uploads? Well it will store them in the webserver's memory if the files are reasonable small otherwise in a temporary location (as returned by `tempfile.gettempdir()`). But how do you specify the maximum file size after which an upload is aborted? By default Flask will happily accept file uploads to an unlimited amount of memory, but you can limit that by setting the `MAX_CONTENT_LENGTH` config key:

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

The code above will limited the maximum allowed payload to 16 megabytes. If a larger file is transmitted, Flask will raise an `RequestEntityTooLarge` exception.

This feature was added in Flask 0.6 but can be achieved in older versions as well by subclassing the request object. For more information on that consult the Werkzeug documentation on file handling.

14.9.3 Upload Progress Bars

A while ago many developers had the idea to read the incoming file in small chunks and store the upload progress in the database to be able to poll the progress with JavaScript from the client. Long story short: the client asks the server every 5 seconds how much it has transmitted already. Do you realize the irony? The client is asking for something it should already know.

Now there are better solutions to that work faster and more reliable. The web changed a lot lately and you can use HTML5, Java, Silverlight or Flash to get a nicer uploading experience on the client side. Look at the following libraries for some nice examples how to do that:

- [Plupload](#) - HTML5, Java, Flash
- [SWFUpload](#) - Flash
- [JumpLoader](#) - Java

14.9.4 An Easier Solution

Because the common pattern for file uploads exists almost unchanged in all applications dealing with uploads, there is a Flask extension called [Flask-Uploads](#) that implements a full fledged upload mechanism with white and blacklisting of extensions and more.

14.10 Caching

When your application runs slow, throw some caches in. Well, at least it's the easiest way to speed up things. What does a cache do? Say you have a function that takes some time to complete but the results would still be good enough if they were 5 minutes old. So then the idea is that you actually put the result of that calculation into a cache for some time.

Flask itself does not provide caching for you, but Werkzeug, one of the libraries it is based on, has some very basic cache support. It supports multiple cache backends, normally you want to use a memcached server.

14.10.1 Setting up a Cache

You create a cache object once and keep it around, similar to how Flask objects are created. If you are using the development server you can create a `SimpleCache` object, that one is a simple cache that keeps the item stored in the memory of the Python interpreter:

```
from werkzeug.contrib.cache import SimpleCache
cache = SimpleCache()
```

If you want to use memcached, make sure to have one of the memcache modules supported (you get them from [PyPI](#)) and a memcached server running somewhere. This is how you connect to such an memcached server then:

```
from werkzeug.contrib.cache import MemcachedCache
cache = MemcachedCache(['127.0.0.1:11211'])
```

If you are using App Engine, you can connect to the App Engine memcache server easily:

```
from werkzeug.contrib.cache import GAEMemcachedCache
cache = GAEMemcachedCache()
```

14.10.2 Using a Cache

Now how can one use such a cache? There are two very important operations: `get()` and `set()`. This is how to use them:

To get an item from the cache call `get()` with a string as key name. If something is in the cache, it is returned. Otherwise that function will return *None*:

```
rv = cache.get('my-item')
```

To add items to the cache, use the `set()` method instead. The first argument is the key and the second the value that should be set. Also a timeout can be provided after which the cache will automatically remove item.

Here a full example how this looks like normally:

```
def get_my_item():
    rv = cache.get('my-item')
    if rv is None:
        rv = calculate_value()
        cache.set('my-item', rv, timeout=5 * 60)
    return rv
```

14.11 View Decorators

Python has a really interesting feature called function decorators. This allow some really neat things for web applications. Because each view in Flask is a function decorators can be used to inject additional functionality to one or more functions. The `route()` decorator is the one you probably used already. But there are use cases for implementing your own decorator. For instance, imagine you have a view that should only be used by people that are logged in to. If a user goes to the site and is not logged in, they should be redirected to the login page. This is a good example of a use case where a decorator is an excellent solution.

14.11.1 Login Required Decorator

So let's implement such a decorator. A decorator is a function that returns a function. Pretty simple actually. The only thing you have to keep in mind when implementing something like this is to update the `__name__`, `__module__` and some other attributes of a function. This is often forgotten, but you don't have to do that by hand, there is a function for that that is used like a decorator (`functools.wraps()`).

This example assumes that the login page is called 'login' and that the current user is stored as `g.user` and `None` if there is no-one logged in:

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

So how would you use that decorator now? Apply it as innermost decorator to a view function. When applying further decorators, always remember that the `route()` decorator is the outermost:

```
@app.route('/secret_page')
@login_required
```

```
def secret_page():  
    pass
```

14.11.2 Caching Decorator

Imagine you have a view function that does an expensive calculation and because of that you would like to cache the generated results for a certain amount of time. A decorator would be nice for that. We're assuming you have set up a cache like mentioned in *Caching*.

Here an example cache function. It generates the cache key from a specific prefix (actually a format string) and the current path of the request. Notice that we are using a function that first creates the decorator that then decorates the function. Sounds awful? Unfortunately it is a little bit more complex, but the code should still be straightforward to read.

The decorated function will then work as follows

1. get the unique cache key for the current request base on the current path.
2. get the value for that key from the cache. If the cache returned something we will return that value.
3. otherwise the original function is called and the return value is stored in the cache for the timeout provided (by default 5 minutes).

Here the code:

```
from functools import wraps  
from flask import request  
  
def cached(timeout=5 * 60, key='view/%s'):  
    def decorator(f):  
        @wraps(f)  
        def decorated_function(*args, **kwargs):  
            cache_key = key % request.path  
            rv = cache.get(cache_key)  
            if rv is not None:  
                return rv  
            rv = f(*args, **kwargs)  
            cache.set(cache_key, rv, timeout=timeout)  
            return rv  
        return decorated_function  
    return decorator
```

Notice that this assumes an instantiated *cache* object is available, see *Caching* for more information.

14.11.3 Templating Decorator

A common pattern invented by the TurboGears guys a while back is a templating decorator. The idea of that decorator is that you return a dictionary with the values passed to the template from the view function and the template is automatically rendered. With that, the following three examples do exactly the same:

```
@app.route('/')
def index():
    return render_template('index.html', value=42)
```

```
@app.route('/')
@templated('index.html')
def index():
    return dict(value=42)
```

```
@app.route('/')
@templated()
def index():
    return dict(value=42)
```

As you can see, if no template name is provided it will use the endpoint of the URL map with dots converted to slashes + '.html'. Otherwise the provided template name is used. When the decorated function returns, the dictionary returned is passed to the template rendering function. If *None* is returned, an empty dictionary is assumed, if something else than a dictionary is returned we return it from the function unchanged. That way you can still use the redirect function or return simple strings.

Here the code for that decorator:

```
from functools import wraps
from flask import request

def templated(template=None):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            template_name = template
            if template_name is None:
                template_name = request.endpoint \
                    .replace('.', '/') + '.html'
            ctx = f(*args, **kwargs)
            if ctx is None:
                ctx = {}
            elif not isinstance(ctx, dict):
                return ctx
            return render_template(template_name, **ctx)
        return decorated_function
    return decorator
```

14.11.4 Endpoint Decorator

When you want to use the werkzeug routing system for more flexibility you need to map the endpoint as defined in the [Rule](#) to a view function. This is possible with this decorator. For example:

```
from flask import Flask
from werkzeug.routing import Rule

app = Flask(__name__)
app.url_map.add(Rule('/', endpoint='index'))

@app.endpoint('index')
def my_index():
    return "Hello world"
```

14.12 Form Validation with WTForms

When you have to work with form data submitted by a browser view code quickly becomes very hard to read. There are libraries out there designed to make this process easier to manage. One of them is [WTForms](#) which we will handle here. If you find yourself in the situation of having many forms, you might want to give it a try.

When you are working with WTForms you have to define your forms as classes first. I recommend breaking up the application into multiple modules (*Larger Applications*) for that and adding a separate module for the forms.

Getting most of WTForms with an Extension

The [Flask-WTF](#) extension expands on this pattern and adds a few handful little helpers that make working with forms and Flask more fun. You can get it from [PyPI](#).

14.12.1 The Forms

This is an example form for a typical registration page:

```
from wtforms import Form, BooleanField, TextField, validators

class RegistrationForm(Form):
    username = TextField('Username', [validators.Length(min=4, max=25)])
    email = TextField('Email Address', [validators.Length(min=6, max=35)])
    password = PasswordField('New Password', [
        validators.Required(),
        validators.EqualTo('confirm', message='Passwords must match')
    ])
    confirm = PasswordField('Repeat Password')
    accept_tos = BooleanField('I accept the TOS', [validators.Required()])
```

14.12.2 In the View

In the view function, the usage of this form looks like this:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(form.username.data, form.email.data,
                    form.password.data)
        db_session.add(user)
        flash('Thanks for registering')
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

Notice that we are implying that the view is using SQLAlchemy here (*SQLAlchemy in Flask*) but this is no requirement of course. Adapt the code as necessary.

Things to remember:

1. create the form from the request form value if the data is submitted via the HTTP *POST* method and args if the data is submitted as *GET*.
2. to validate the data, call the `validate()` method which will return *True* if the data validates, *False* otherwise.
3. to access individual values from the form, access `form.<NAME>.data`.

14.12.3 Forms in Templates

Now to the template side. When you pass the form to the templates you can easily render them there. Look at the following example template to see how easy this is. WTForms does half the form generation for us already. To make it even nicer, we can write a macro that renders a field with label and a list of errors if there are any.

Here's an example `_formhelpers.html` template with such a macro:

```
{% macro render_field(field) %}
<dt>{{ field.label }}
<dd>{{ field(**kwargs)|safe }}
{% if field.errors %}
<ul class="errors">
    {% for error in field.errors %}<li>{{ error }}{% endfor %}
</ul>
{% endif %}
</dd>
{% endmacro %}
```

This macro accepts a couple of keyword arguments that are forwarded to WTForm's field function that renders the field for us. The keyword arguments will be inserted as HTML attributes. So for example you can call `render_field(form.username, class='username')` to add a class to the input element. Note that WTForms returns

standard Python unicode strings, so we have to tell Jinja2 that this data is already HTML escaped with the `|safe` filter.

Here the `register.html` template for the function we used above which takes advantage of the `_formhelpers.html` template:

```
{% from "_formhelpers.html" import render_field %}
<form method="post" action="/register">
  <dl>
    {{ render_field(form.username) }}
    {{ render_field(form.email) }}
    {{ render_field(form.password) }}
    {{ render_field(form.confirm) }}
    {{ render_field(form.accept_tos) }}
  </dl>
  <p><input type=submit value=Register>
</form>
```

For more information about WTForms, head over to the [WTForms website](#).

14.13 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.

14.13.1 Base Template

This template, which we’ll call `layout.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It’s the job of “child” templates to fill the empty blocks with content:

```
<!doctype html>
<html>
  <head>
    {% block head %}
      <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
      <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
        &copy; Copyright 2010 by <a href="http://domain.invalid/">you</a>.
      {% endblock %}
    </div>
  </body>
</html>
```

```
</div>
</body>
```

In this example, the `{% block %}` tags define four blocks that child templates can fill in. All the *block* tag does is tell the template engine that a child template may override those portions of the template.

14.13.2 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.
    </p>
{% 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() }}`.

14.14 Message Flashing

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

14.14.1 Simple Flashing

So here is 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 successfully 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 %}

```

14.14.2 Flashing With Categories

New in version 0.3. It is also possible to provide categories when flashing a message. The default category if nothing is provided is 'message'. Alternative categories can be used to give the user better feedback. For example error messages could be displayed with a red background.

To flash a message with a different category, just use the second argument to the flash() function:

```
flash(u'Invalid password provided', 'error')
```

Inside the template you then have to tell the get_flashed_messages() function to also return the categories. The loop looks slightly different in that situation then:

```

{% with messages = get_flashed_messages(with_categories=true) %}
    {% if messages %}
        <ul class=flashes>
            {% for category, message in messages %}
                <li class="{{ category }}">{{ message }}</li>
            {% endfor %}
        </ul>
    {% endif %}
{% endwith %}

```

This is just one example of how to render these flashed messages. One might also use the category to add a prefix such as Error: to the message.

14.15 AJAX with jQuery

[jQuery](#) is a small JavaScript library commonly used to simplify working with the DOM and JavaScript in general. It is the perfect tool to make web applications more dynamic by exchanging JSON between server and client.

JSON itself is a very lightweight transport format, very similar to how Python primitives (numbers, strings, dicts and lists) look like which is widely supported and very easy to parse. It became popular a few years ago and quickly replaced XML as transport format in web applications.

If you have Python 2.6 JSON will work out of the box, in Python 2.5 you will have to install the [simplejson](#) library from PyPI.

14.15.1 Loading jQuery

In order to use jQuery, you have to download it first and place it in the static folder of your application and then ensure it's loaded. Ideally you have a layout template that is used for all pages where you just have to add a script statement to the bottom of your `<body>` to load jQuery:

```
<script type=text/javascript src="{{
    url_for('static', filename='jquery.js') }}"></script>
```

Another method is using Google's [AJAX Libraries API](#) to load jQuery:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.js"></script>
<script>window.jQuery || document.write('<script src="{{
    url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

In this case you have to put jQuery into your static folder as a fallback, but it will first try to load it directly from Google. This has the advantage that your website will probably load faster for users if they went to at least one other website before using the same jQuery version from Google because it will already be in the browser cache.

14.15.2 Where is My Site?

Do you know where your application is? If you are developing the answer is quite simple: it's on localhost port something and directly on the root of that server. But what if you later decide to move your application to a different location? For example to `http://example.com/myapp`? On the server side this never was a problem because we were using the handy `url_for()` function that could answer that question for us, but if we are using jQuery we should not hardcode the path to the application but make that dynamic, so how can we do that?

A simple method would be to add a script tag to our page that sets a global variable to the prefix to the root of the application. Something like this:

```
<script type=text/javascript>
    $SCRIPT_ROOT = {{ request.script_root|tojson|safe }};
</script>
```

The `|safe` is necessary so that Jinja does not escape the JSON encoded string with HTML rules. Usually this would be necessary, but we are inside a *script* block here where different rules apply.

Information for Pros

In HTML the *script* tag is declared *CDATA* which means that entities will not be parsed. Everything until `</script>` is handled as script. This also means that there must never be any `</` between the script tags. `|tojson` is kind enough to do the right thing here and escape slashes for you (`{{ "</script>"|tojson|safe }}` is rendered as `"<\script>"`).

14.15.3 JSON View Functions

Now let's create a server side function that accepts two URL arguments of numbers which should be added together and then sent back to the application in a JSON object. This is a really ridiculous example and is something you usually would do on the client side alone, but a simple example that shows how you would use jQuery and Flask nonetheless:

```
from flask import Flask, jsonify, render_template, request
app = Flask(__name__)

@app.route('/_add_numbers')
def add_numbers():
    a = request.args.get('a', 0, type=int)
    b = request.args.get('b', 0, type=int)
    return jsonify(result=a + b)

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

As you can see I also added an *index* method here that renders a template. This template will load jQuery as above and have a little form we can add two numbers and a link to trigger the function on the server side.

Note that we are using the `get()` method here which will never fail. If the key is missing a default value (here 0) is returned. Furthermore it can convert values to a specific type (like in our case *int*). This is especially handy for code that is triggered by a script (APIs, JavaScript etc.) because you don't need special error reporting in that case.

14.15.4 The HTML

Your *index.html* template either has to extend a *layout.html* template with jQuery loaded and the `$SCRIPT_ROOT` variable set, or do that on the top. Here's the HTML code needed for our little application (*index.html*). Notice that we also drop the script directly into the HTML here. It is usually a better idea to have that in a separate script file:

```
<script type=text/javascript>
$(function() {
    $('#calculate').bind('click', function() {
        $.getJSON($SCRIPT_ROOT + '/_add_numbers', {
            a: $('#input[name="a"]').val(),
            b: $('#input[name="b"]').val()
        }, function(data) {
            $("#result").text(data.result);
        });
    });
    return false;
});
```

```

});
</script>
<h1>jQuery Example</h1>
<p><input type=text size=5 name=a> +
    <input type=text size=5 name=b> =
    <span id=result>?</span>
<p><a href=# id=calculate>calculate server side</a>

```

I won't get into detail here about how jQuery works, just a very quick explanation of the little bit of code above:

1. `$(function() { ... })` specifies code that should run once the browser is done loading the basic parts of the page.
2. `$('#selector')` selects an element and lets you operate on it.
3. `element.bind('event', func)` specifies a function that should run when the user clicked on the element. If that function returns *false*, the default behaviour will not kick in (in this case, navigate to the # URL).
4. `$.getJSON(url, data, func)` sends a *GET* request to *url* and will send the contents of the *data* object as query parameters. Once the data arrived, it will call the given function with the return value as argument. Note that we can use the `$SCRIPT_ROOT` variable here that we set earlier.

If you don't get the whole picture, download the [sourcecode for this example](#) from github.

14.16 Custom Error Pages

Flask comes with a handy `abort()` function that aborts a request with an HTTP error code early. It will also provide a plain black and white error page for you with a basic description, but nothing fancy.

Depending on the error code it is less or more likely for the user to actually see such an error.

14.16.1 Common Error Codes

The following error codes are some that are often displayed to the user, even if the application behaves correctly:

404 Not Found The good old “chap, you made a mistake typing that URL” message. So common that even novices to the internet know that 404 means: damn, the thing I was looking for is not there. It's a very good idea to make sure there is actually something useful on a 404 page, at least a link back to the index.

403 Forbidden If you have some kind of access control on your website, you will have to send a 403 code for disallowed resources. So make sure the user is not lost when they try to access a forbidden resource.

410 Gone Did you know that there the “404 Not Found” has a brother named “410 Gone”? Few people actually implement that, but the idea is that resources that previously existed and got deleted answer with 410 instead of 404. If you are not deleting documents permanently from the database but just mark them as deleted, do the user a favour and use the 410 code instead and display a message that what they were looking for was deleted for all eternity.

500 Internal Server Error Usually happens on programming errors or if the server is overloaded. A terrible good idea to have a nice page there, because your application *will* fail sooner or later (see also: *Handling Application Errors*).

14.16.2 Error Handlers

An error handler is a function, just like a view function, but it is called when an error happens and is passed that error. The error is most likely a `HTTPException`, but in one case it can be a different error: a handler for internal server errors will be passed other exception instances as well if they are uncaught.

An error handler is registered with the `errorhandler()` decorator and the error code of the exception. Keep in mind that Flask will *not* set the error code for you, so make sure to also provide the HTTP status code when returning a response.

Here an example implementation for a “404 Page Not Found” exception:

```
from flask import render_template

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

An example template might be this:

```
{% extends "layout.html" %}
{% block title %}Page Not Found{% endblock %}
{% block body %}
    <h1>Page Not Found</h1>
    <p>What you were looking for is just not there.
    <p><a href="{{ url_for('index') }}">go somewhere nice</a>
{% endblock %}
```

14.17 Lazily Loading Views

Flask is usually used with the decorators. Decorators are simple and you have the URL right next to the function that is called for that specific URL. However there is a downside to this approach: it means all your code that uses decorators has to be imported upfront or Flask will never actually find your function.

This can be a problem if your application has to import quick. It might have to do that on systems like Google’s App Engine or other systems. So if you suddenly notice

that your application outgrows this approach you can fall back to a centralized URL mapping.

The system that enables having a central URL map is the `add_url_rule()` function. Instead of using decorators, you have a file that sets up the application with all URLs.

14.17.1 Converting to Centralized URL Map

Imagine the current application looks somewhat like this:

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

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

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

Then the centralized approach you would have one file with the views (*views.py*) but without any decorator:

```
def index():
    pass

def user(username):
    pass
```

And then a file that sets up an application which maps the functions to URLs:

```
from flask import Flask
from yourapplication import views
app = Flask(__name__)
app.add_url_rule('/', view_func=views.index)
app.add_url_rule('/user/<username>', view_func=views.user)
```

14.17.2 Loading Late

So far we only split up the views and the routing, but the module is still loaded upfront. The trick to actually load the view function as needed. This can be accomplished with a helper class that behaves just like a function but internally imports the real function on first use:

```
from werkzeug import import_string, cached_property

class LazyView(object):

    def __init__(self, import_name):
```

```

self.__module__, self.__name__ = import_name.rsplit('.', 1)
self.import_name = import_name

@cached_property
def view(self):
    return import_string(self.import_name)

def __call__(self, *args, **kwargs):
    return self.view(*args, **kwargs)

```

What's important here is that `__module__` and `__name__` are properly set. This is used by Flask internally to figure out how to name the URL rules in case you don't provide a name for the rule yourself.

Then you can define your central place to combine the views like this:

```

from flask import Flask
from yourapplication.helpers import LazyView
app = Flask(__name__)
app.add_url_rule('/',
                 view_func=LazyView('yourapplication.views.index'))
app.add_url_rule('/user/<username>',
                 view_func=LazyView('yourapplication.views.user'))

```

You can further optimize this in terms of amount of keystrokes needed to write this by having a function that calls into `add_url_rule()` by prefixing a string with the project name and a dot, and by wrapping `view_func` in a `LazyView` as needed:

```

def url(url_rule, import_name, **options):
    view = LazyView('yourapplication.' + import_name)
    app.add_url_rule(url_rule, view_func=view, **options)

url('/', 'views.index')
url('/user/<username>', 'views.user')

```

One thing to keep in mind is that before and after request handlers have to be in a file that is imported upfront to work properly on the first request. The same goes for any kind of remaining decorator.

14.18 MongoKit in Flask

Using a document database rather than a full DBMS gets more common these days. This pattern shows how to use MongoKit, a document mapper library, to integrate with MongoDB.

This pattern requires a running MongoDB server and the MongoKit library installed. There are two very common ways to use MongoKit. I will outline each of them here:

14.18.1 Declarative

The default behaviour of MongoKit is the declarative one that is based on common ideas from Django or the SQLAlchemy declarative extension.

Here an example *app.py* module for your application:

```
from flask import Flask
from mongokit import Connection, Document

# configuration
MONGODB_HOST = 'localhost'
MONGODB_PORT = 27017

# create the little application object
app = Flask(__name__)
app.config.from_object(__name__)

# connect to the database
connection = Connection(app.config['MONGODB_HOST'],
                        app.config['MONGODB_PORT'])
```

To define your models, just subclass the *Document* class that is imported from MongoKit. If you've seen the SQLAlchemy pattern you may wonder why we do not have a session and even do not define a *init_db* function here. On the one hand, MongoKit does not have something like a session. This sometimes makes it more to type but also makes it blazingly fast. On the other hand, MongoDB is schemaless. This means you can modify the data structure from one insert query to the next without any problem. MongoKit is just schemaless too, but implements some validation to ensure data integrity.

Here is an example document (put this also into *app.py*, e.g.):

```
def max_length(length):
    def validate(value):
        if len(value) <= length:
            return True
        raise Exception('%s must be at most %s characters long' % length)
    return validate

class User(Document):
    structure = {
        'name': unicode,
        'email': unicode,
    }
    validators = {
        'name': max_length(50),
        'email': max_length(120)
    }
    use_dot_notation = True
    def __repr__(self):
        return '<User %r>' % (self.name)
```

```
# register the User document with our current connection
connection.register([User])
```

This example shows you how to define your schema (named structure), a validator for the maximum character length and uses a special MongoKit feature called *use_dot_notation*. Per default MongoKit behaves like a python dictionary but with *use_dot_notation* set to *True* you can use your documents like you use models in nearly any other ORM by using dots to separate between attributes.

You can insert entries into the database like this:

```
>>> from yourapplication.database import connection
>>> from yourapplication.models import User
>>> collection = connection['test'].users
>>> user = collection.User()
>>> user['name'] = u'admin'
>>> user['email'] = u'admin@localhost'
>>> user.save()
```

Note that MongoKit is kinda strict with used column types, you must not use a common *str* type for either *name* or *email* but unicode.

Querying is simple as well:

```
>>> list(collection.User.find())
[<User u'admin'>]
>>> collection.User.find_one({'name': u'admin'})
<User u'admin'>
```

14.18.2 PyMongo Compatibility Layer

If you just want to use PyMongo, you can do that with MongoKit as well. You may use this process if you need the best performance to get. Note that this example does not show how to couple it with Flask, see the above MongoKit code for examples:

```
from MongoKit import Connection
```

```
connection = Connection()
```

To insert data you can use the *insert* method. We have to get a collection first, this is somewhat the same as a table in the SQL world.

```
>>> collection = connection['test'].users
>>> user = {'name': u'admin', 'email': u'admin@localhost'}
>>> collection.insert(user)
```

```
print list(collection.find()) print collection.find_one({'name': u'admin'})
```

MongoKit will automatically commit for us.

To query your database, you use the collection directly:

```
>>> list(collection.find())
[{'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u'admin@localhost'}]
>>> collection.find_one({'name': u'admin'})
{'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u'admin@localhost'}
```

These results are also dict-like objects:

```
>>> r = collection.find_one({'name': u'admin'})
>>> r['email']
u'admin@localhost'
```

For more information about MongoKit, head over to the [website](#).

14.19 Adding a favicon

A “favicon” is an icon used by browsers for tabs and bookmarks. This helps to distinguish your website and to give it a unique brand.

A common question is how to add a favicon to a flask application. First, of course, you need an icon. It should be 16 × 16 pixels and in the ICO file format. This is not a requirement but a de-facto standard supported by all relevant browsers. Put the icon in your static directory as `favicon.ico`.

Now, to get browsers to find your icon, the correct way is to add a link tag in your HTML. So, for example:

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

That’s all you need for most browsers, however some really old ones do not support this standard. The old de-facto standard is to serve this file, with this name, at the website root. If your application is not mounted at the root path of the domain you either need to configure the webserver to serve the icon at the root or if you can’t do that you’re out of luck. If however your application is the root you can simply route a redirect:

```
app.add_url_rule('/favicon.ico',
                 redirect_to=url_for('static', filename='favicon.ico'))
```

If you want to save the extra redirect request you can also write a view using `send_from_directory()`:

```
import os
from flask import send_from_directory

@app.route('/favicon.ico')
def favicon():
    return send_from_directory(os.path.join(app.root_path, 'static'),
                              'favicon.ico', mimetype='image/vnd.microsoft.icon')
```

We can leave out the explicit mimetype and it will be guessed, but we may as well specify it to avoid the extra guessing, as it will always be the same.

The above will serve the icon via your application and if possible it's better to configure your dedicated web server to serve it; refer to the webserver's documentation.

14.19.1 See also

- The [Favicon](#) article on Wikipedia

14.20 Streaming Contents

Sometimes you want to send an enormous amount of data to the client, much more than you want to keep in memory. When you are generating the data on the fly though, how do you send that back to the client without the roundtrip to the filesystem?

The answer is by using generators and direct responses.

14.20.1 Basic Usage

This is a basic view function that generates a lot of CSV data on the fly. The trick is to have an inner function that uses a generator to generate data and to then invoke that function and pass it to a response object that has the `direct_passthrough` flag set. This flag is used to inform the system that data is generated on the fly and should be passed through without buffering:

```
from flask import Response

@app.route('/large.csv')
def generate_large_csv():
    def generate():
        for row in iter_all_rows():
            yield ','.join(row) + '\n'
    return Response(generate(), direct_passthrough=True,
                    mimetype='text/csv')
```

Each yield expression is directly sent to the browser. Now though that some WSGI middlewares might break streaming, so be careful there in debug environments with profilers and other things you might have enabled.

14.20.2 Streaming from Templates

The Jinja2 template engine also supports rendering templates piece by piece. This functionality is not directly exposed by Flask because it is quite uncommon, but you can easily do it yourself:

```
from flask import Response

def stream_template(template_name, **context):
```

```

app.update_template_context(context)
t = app.jinja_env.get_template(template_name)
rv = t.stream(context)
rv.enable_buffering(5)
return rv

@app.route('/my-large-page.html')
def render_large_template():
    rows = iter_all_rows()
    return Response(stream_template('the_template.html', rows=rows),
                    direct_passthrough=True)

```

The trick here is to get the template object from the Jinja2 environment on the application and to call `stream()` instead of `render()` which returns a stream object instead of a string. Since we're bypassing the Flask template render functions and using the template object itself we have to make sure to update the render context ourselves by calling `update_template_context()`. The template is then evaluated as the stream is iterated over. Since each time you do a `yield` the server will flush the content to the client you might want to buffer up a few items in the template which you can do with `rv.enable_buffering(size)`. 5 is a sane default.

DEPLOYMENT OPTIONS

Depending on what you have available there are multiple ways to run Flask applications. You can use the builtin server during development, but you should use a full deployment option for production applications. (Do not use the builtin development server in production.) Several options are available and documented here.

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 Flask application object is the actual WSGI application.

15.1 `mod_wsgi` (Apache)

If you are using the [Apache](#) webserver, consider using `mod_wsgi`.

Watch Out

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to `mod_wsgi`.

15.1.1 Installing *mod_wsgi*

If you don't have *mod_wsgi* installed yet you have to either install it using a package manager or compile it yourself. The `mod_wsgi` [installation instructions](#) cover source installations on UNIX systems.

If you are using Ubuntu/Debian you can `apt-get` it and activate it as follows:

```
# apt-get install libapache2-mod-wsgi
```

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.

15.1.2 Creating a `.wsgi` file

To run your application you need a `yourapplication.wsgi` file. This file contains the code `mod_wsgi` is executing on startup to get the application object. The object called `application` in that file is then used as application.

For most applications the following file should be sufficient:

```
from yourapplication import app as application
```

If you don't have a factory function for application creation but a singleton instance you can directly import that one as `application`.

Store that file somewhere that you will find it again (e.g.: `/var/www/yourapplication`) and make sure that `yourapplication` and all the libraries that are in use are on the python load path. If you don't want to install it system wide consider using a [virtual python](#) instance.

15.1.3 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.

15.1.4 Troubleshooting

If your application does not run, follow this guide to troubleshoot:

Problem: application does not run, errorlog shows `SystemExit ignored` You have a `app.run()` call in your application file that is not guarded by an `if __name__ == '__main__':` condition. Either remove that `run()` call from the file and move it into a separate *run.py* file or put it into such an if block.

Problem: application gives permission errors Probably caused by your application running as the wrong user. Make sure the folders the application needs access to have the proper privileges set and the application runs as the correct user (user and group parameter to the *WSGIDaemonProcess* directive)

Problem: application dies with an error on print Keep in mind that `mod_wsgi` disallows doing anything with `sys.stdout` and `sys.stderr`. You can disable this protection from the config by setting the *WSGIRestrictStdout* to off:

```
WSGIRestrictStdout Off
```

Alternatively you can also replace the standard out in the *.wsgi* file with a different stream:

```
import sys
sys.stdout = sys.stderr
```

Problem: accessing resources gives IO errors Your application probably is a single *.py* file you symlinked into the site-packages folder. Please be aware that this does not work, instead you either have to put the folder into the `pythonpath` the file is stored in, or convert your application into a package.

The reason for this is that for non-installed packages, the module filename is used to locate the resources and for symlinks the wrong filename is picked up.

15.1.5 Support for Automatic Reloading

To help deployment tools you can activate support for automatic reloading. Whenever something changes the *.wsgi* file, *mod_wsgi* will reload all the daemon processes for us.

For that, just add the following directive to your *Directory* section:

```
WSGIScriptReloading On
```

15.1.6 Working with Virtual Environments

Virtual environments have the advantage that they never install the required dependencies system wide so you have a better control over what is used where. If you want to use a virtual environment with *mod_wsgi* you have to modify your *.wsgi* file slightly.

Add the following lines to the top of your *.wsgi* file:

```
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

This sets up the load paths according to the settings of the virtual environment. Keep in mind that the path has to be absolute.

15.2 CGI

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

This is also the way you can use a Flask application on Google's [App Engine](#), where execution happens in a CGI-like environment.

Watch Out

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to CGI / app engine.

15.2.1 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)
```

15.2.2 Server Setup

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

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

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

For more information consult the documentation of your webserver.

15.3 FastCGI

FastCGI is a deployment option on servers like [nginx](#), [lighttpd](#), and [cherokee](#); see *uWSGI* and *Other Servers* for other options. To use your WSGI application with any of them you will need a FastCGI server first. The most popular one is [flup](#) which we will use for this guide. Make sure to have it installed to follow along.

Watch Out

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to FastCGI.

15.3.1 Creating a *.fcgi* file

First you need to create the FastCGI server file. Let's call it *yourapplication.fcgi*:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from yourapplication import app

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

This is enough for Apache to work, however [nginx](#) and older versions of [lighttpd](#) need a socket to be explicitly passed to communicate with the FastCGI server. For that to work you need to pass the path to the socket to the `WSGIServer`:

```
WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()
```

The path has to be the exact same path you define in the server config.

Save the *yourapplication.fcgi* file somewhere you will find it again. It makes sense to have that in */var/www/yourapplication* or something similar.

Make sure to set the executable bit on that file so that the servers can execute it:

```
# chmod +x /var/www/yourapplication/yourapplication.fcgi
```

15.3.2 Configuring [lighttpd](#)

A basic FastCGI configuration for [lighttpd](#) looks like that:

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

```

        "check-local" => "disable",
        "max-procs" => 1
    ))
)

alias.url = (
    "/static/" => "/path/to/your/static"
)

url.rewrite-once = (
    "^(/static.*)$" => "$1",
    "^(/.*)$" => "/yourapplication.fcgi$1"
)

```

Remember to enable the FastCGI, alias and rewrite modules. This configuration binds the application to */yourapplication*. If you want the application to work in the URL root you have to work around a `lighttpd` bug with the `LighttpdCGIRootFix` middleware.

Make sure to apply it only if you are mounting the application the URL root. Also, see the `Lighty` docs for more information on `FastCGI` and `Python` (note that explicitly passing a socket to `run()` is no longer necessary).

15.3.3 Configuring nginx

Installing FastCGI applications on `nginx` is a bit different because by default no FastCGI parameters are forwarded.

A basic flask FastCGI configuration for `nginx` looks like this:

```

location = /yourapplication { rewrite ^ /yourapplication/ last; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
    include fastcgi_params;
    fastcgi_split_path_info ^(/yourapplication)(.*)$;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}

```

This configuration binds the application to */yourapplication*. If you want to have it in the URL root it's a bit simpler because you don't have to figure out how to calculate `PATH_INFO` and `SCRIPT_NAME`:

```

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

```

15.3.4 Running FastCGI Processes

Since Nginx and others do not load FastCGI apps, you have to do it by yourself. [Supervisor can manage FastCGI processes](#). You can look around for other FastCGI process managers or write a script to run your `.fcgi` file at boot, e.g. using a SysV `init.d` script. For a temporary solution, you can always run the `.fcgi` script inside GNU screen. See `man screen` for details, and note that this is a manual solution which does not persist across system restart:

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

15.3.5 Debugging

FastCGI deployments tend to be hard to debug on most web servers. Very often the only thing the server log tells you is something along the lines of “premature end of headers”. In order to debug the application the only thing that can really give you ideas why it breaks is switching to the correct user and executing the application by hand.

This example assumes your application is called `application.fcgi` and that your web-server user is `www-data`:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcgi", line 4, in <module>
ImportError: No module named yourapplication
```

In this case the error seems to be “yourapplication” not being on the python path. Common problems are:

- Relative paths being used. Don’t rely on the current working directory
- The code depending on environment variables that are not set by the web server.
- Different python interpreters being used.

15.4 uWSGI

uWSGI is a deployment option on servers like [nginx](#), [lighttpd](#), and [cherokee](#); see *FastCGI* and *Other Servers* for other options. To use your WSGI application with uWSGI protocol you will need a uWSGI server first. uWSGI is both a protocol and an application server; the application server can serve uWSGI, FastCGI, and HTTP protocols.

The most popular uWSGI server is [uwsgi](#), which we will use for this guide. Make sure to have it installed to follow along.

Watch Out

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to uWSGI.

15.4.1 Starting your app with uwsgi

uwsgi is designed to operate on WSGI callables found in python modules.

Given a flask application in `myapp.py`, use the following command:

```
$ uwsgi -s /tmp/uwsgi.sock --module myapp --callable app
```

Or, if you prefer:

```
$ uwsgi -s /tmp/uwsgi.sock myapp:app
```

15.4.2 Configuring nginx

A basic flask uWSGI configuration for nginx looks like this:

```
location = /yourapplication { rewrite ^ /yourapplication/; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
    include uwsgi_params;
    uwsgi_param SCRIPT_NAME /yourapplication;
    uwsgi_modifier1 30;
    uwsgi_pass unix:/tmp/uwsgi.sock;
}
```

This configuration binds the application to */yourapplication*. If you want to have it in the URL root it's a bit simpler because you don't have to tell it the WSGI `SCRIPT_NAME` or set the uwsgi modifier to make use of it:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include uwsgi_params;
    uwsgi_pass unix:/tmp/uwsgi.sock;
}
```

15.5 Other Servers

There are popular servers written in Python that allow the execution of WSGI applications as well. These servers stand alone when they run; you can proxy to them from

your web server.

15.5.1 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()
```

15.5.2 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()
```

15.5.3 Gunicorn

Gunicorn ‘Green Unicorn’ is a WSGI HTTP Server for UNIX. It’s a pre-fork worker model ported from Ruby’s Unicorn project. It supports both **eventlet** and **greenlet**. Running a Flask application on this server is quite simple:

```
gunicorn myproject:app
```

Gunicorn provides many command-line options – see `gunicorn -h`. For example, to run a Flask application with 4 worker processes (`-w 4`) binding to localhost port 4000 (`-b 127.0.0.1:4000`):

```
gunicorn -w 4 -b 127.0.0.1:4000 myproject:app
```

15.5.4 Proxy Setups

If you deploy your application using one of these servers behind an HTTP proxy you will need to rewrite a few headers in order for the application to work. The two problematic values in the WSGI environment usually are *REMOTE_ADDR* and *HTTP_HOST*. Werkzeug ships a fixer that will solve some common setups, but you might want to write your own WSGI middleware for specific setups.

The most common setup invokes the host being set from *X-Forwarded-Host* and the remote address from *X-Forwarded-For*:

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

Please keep in mind that it is a security issue to use such a middleware in a non-proxy setup because it will blindly trust the incoming headers which might be forged by malicious clients.

If you want to rewrite the headers from another header, you might want to use a fixer like this:

```
class CustomProxyFix(object):

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        host = environ.get('HTTP_X_FHOST', '')
        if host:
            environ['HTTP_HOST'] = host
        return self.app(environ, start_response)
```

```
app.wsgi_app = CustomProxyFix(app.wsgi_app)
```

BECOMING BIG

Your application is becoming more and more complex? If you suddenly realize that Flask does things in a way that does not work out for your application there are ways to deal with that.

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 bring those two together. Being a microframework Flask does not do much more than combining existing libraries - there is not a lot of code involved. What that means for large applications is that it's very easy to take the code from Flask and put it into a new module within the applications and expand on that.

Flask is designed to be extended and modified in a couple of different ways:

- Flask extensions. For a lot of reusable functionality you can create extensions. For extensions a number of hooks exist throughout Flask with signals and call-back functions.
- Subclassing. The majority of functionality can be changed by creating a new subclass of the Flask class and overriding methods provided for this exact purpose.
- Forking. If nothing else works out you can just take the Flask codebase at a given point and copy/paste it into your application and change it. Flask is designed with that in mind and makes this incredible easy. You just have to take the package and copy it into your application's code and rename it (for example to *framework*). Then you can start modifying the code in there.

16.1 Why consider Forking?

The majority of code of Flask is within Werkzeug and Jinja2. These libraries do the majority of the work. Flask is just the paste that glues those together. For every project there is the point where the underlying framework gets in the way (due to assumptions the original developers had). This is natural because if this would not be the case, the framework would be a very complex system to begin with which causes a steep learning curve and a lot of user frustration.

This is not unique to Flask. Many people use patched and modified versions of their framework to counter shortcomings. This idea is also reflected in the license of Flask.

You don't have to contribute any changes back if you decide to modify the framework.

The downside of forking is of course that Flask extensions will most likely break because the new framework has a different import name. Furthermore integrating upstream changes can be a complex process, depending on the number of changes. Because of that, forking should be the very last resort.

16.2 Scaling like a Pro

For many web applications the complexity of the code is less an issue than the scaling for the number of users or data entries expected. Flask by itself is only limited in terms of scaling by your application code, the data store you want to use and the Python implementation and webserver you are running on.

Scaling well means for example that if you double the amount of servers you get about twice the performance. Scaling bad means that if you add a new server the application won't perform any better or would not even support a second server.

There is only one limiting factor regarding scaling in Flask which are the context local proxies. They depend on context which in Flask is defined as being either a thread, process or greenlet. If your server uses some kind of concurrency that is not based on threads or greenlets, Flask will no longer be able to support these global proxies. However the majority of servers are using either threads, greenlets or separate processes to achieve concurrency which are all methods well supported by the underlying Werkzeug library.

16.3 Dialogue with the Community

The Flask developers are very interested to keep everybody happy, so as soon as you find an obstacle in your way, caused by Flask, don't hesitate to contact the developers on the mailinglist or IRC channel. The best way for the Flask and Flask-extension developers to improve it for larger applications is getting feedback from users.

Part II

API REFERENCE

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

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.

17.1 Application Object

```
class flask.Flask(import_name, static_path=None, static_url_path=None,  
                 static_folder='static', template_folder='templates')
```

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see `open_resource()`.

Usually you create a Flask instance in your main module or in the `__init__.py` file of your package like this:

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

About the First Parameter

The idea of the first parameter is to give Flask an idea what belongs to your application. This name is used to find resources on the file system, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, `__name__` is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in *yourapplication/app.py* you should create it with one of the two versions below:

```
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with `__name__`, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

New in version 0.5: The `static_path` parameter was added.

Parameters

- **import_name** – the name of the application package
- **static_path** – can be used to specify a different path for the static files on the web. Defaults to `/static`. This does not affect the folder the files are served *from*.

`add_url_rule(*args, **kwargs)`

Connects a URL rule. Works exactly like the `route()` decorator. If a `view_func` is provided it will be registered with the endpoint.

Basically this example:

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

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('/', 'index', index)
```

If the `view_func` is not provided you will need to connect the endpoint to a view function like so:

```
app.view_functions['index'] = index
```

If a view function is provided some defaults can be specified directly on the view function. For more information refer to *View Function Options*. Changed in version 0.2: `view_func` parameter added. Changed in version 0.6: `OPTIONS` is added automatically as method.

Parameters

- **rule** – the URL rule as string

- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **view_func** – the function to call when serving a request to the provided endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. `methods` is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

`after_request(*args, **kwargs)`

Register a function to be run after each request. Your function must take one parameter, a `response_class` object and return a new response object or the same (see `process_response()`).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

`after_request_funcs = None`

A dictionary with lists of functions that should be called after each request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to open database connections or getting hold of the currently logged in user. To register a function here, use the `after_request()` decorator.

`before_first_request(*args, **kwargs)`

Registers a function to be run before the first request to this instance of the application. New in version 0.8.

`before_first_request_funcs = None`

A lists of functions that should be called at the beginning of the first request to this instance. To register a function here, use the `before_first_request()` decorator. New in version 0.8.

`before_request(*args, **kwargs)`

Registers a function to run before each request.

`before_request_funcs = None`

A dictionary with lists of functions that should be called at the beginning of the request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to open database connections or getting hold of the currently logged in user. To register a function here, use the `before_request()` decorator.

`blueprints = None`

all the attached blueprints in a directory by name. Blueprints can be attached multiple times so this dictionary does not tell you how often they got attached. New in version 0.7.

`config = None`

The configuration dictionary as `Config`. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

`context_processor(*args, **kwargs)`

Registers a template context processor function.

`create_global_jinja_loader()`

Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the `create_jinja_loader()` function instead.

The global loader dispatches between the loaders of the application and the individual blueprints. New in version 0.7.

`create_jinja_environment()`

Creates the Jinja2 environment based on `jinja_options` and `select_jinja_autoescape()`. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior. New in version 0.5.

`create_url_adapter(request)`

Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly. New in version 0.6.

`debug`

The debug flag. Set this to *True* to enable debugging of the application. In debug mode the debugger will kick in when an unhandled exception occurs and the integrated server will automatically reload the application if changes in the code are detected.

This attribute can also be configured from the config with the *DEBUG* configuration key. Defaults to *False*.

`debug_log_format = '_____\\n%(levelname)s`

The logging format used for the debug logger. This is only used when the application is in debug mode, otherwise the attached logging handler does the formatting. New in version 0.3.

`default_config = ImmutableDict({'USE_X_SENDFILE': False, 'PROPAGATE_EXCEPTIONS': True})`

Default configuration parameters.

`dispatch_request()`

Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call `make_response()`. Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

`do_teardown_request()`

Called after the actual request dispatching and will call every as `teardown_request()` decorated function. This is not actually called by

the Flask object itself but is always triggered when the request context is popped. That way we have a tighter control over certain resources under testing environments.

enable_modules = True

Enable the deprecated module support? This is active by default in 0.7 but will be changed to False in 0.8. With Flask 1.0 modules will be removed in favor of Blueprints

endpoint(*args, **kwargs)

A decorator to register a function as an endpoint. Example:

```
@app.endpoint('example.endpoint')
def example():
    return "example"
```

Parameters endpoint – the name of the endpoint

error_handler_spec = None

A dictionary of all registered error handlers. The key is *None* for error handlers active on the application, otherwise the key is the name of the blueprint. Each key points to another dictionary where the key is the status code of the http exception. The special key *None* points to a list of tuples where the first item is the class for the instance check and the second the error handler function.

To register a error handler, use the `errorhandler()` decorator.

errorhandler(*args, **kwargs)

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

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

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

You can also register a function as error handler without using the `errorhandler()` decorator. The following example is equivalent to the one above:

```
def page_not_found(error):
    return 'This page does not exist', 404
app.error_handler_spec[None][404] = page_not_found
```

Setting error handlers via assignments to `error_handler_spec` however is discouraged as it requires fiddling with nested dictionaries and the special case for arbitrary exception types.

The first *None* refers to the active blueprint. If the error handler should be application wide *None* shall be used. New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

Parameters code – the code as integer for the handler

extensions = None

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things. For backwards compatibility extensions should register themselves like this:

```
if not hasattr(app, 'extensions'):
    app.extensions = {}
app.extensions['extensionname'] = SomeObject()
```

The key must match the name of the *flaskext* module. For example in case of a “Flask-Foo” extension in *flaskext.foo*, the key would be `'foo'`. New in version 0.7.

full_dispatch_request()

Dispatches the request and on top of that performs request pre and post-processing as well as HTTP exception catching and error handling. New in version 0.7.

got_first_request

This attribute is set to *True* if the application started handling the first request. New in version 0.8.

handle_exception(*e*)

Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

handle_http_exception(*e*)

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

handle_user_exception(*e*)

This method is called whenever an exception occurs that should be handled. A special case are `HTTPExceptions` which are forwarded by this function to the `handle_http_exception()` method. This function will either return a response value or reraise the exception with the same traceback. New in version 0.7.

has_static_folder

This is *True* if the package bound object's container has a folder named `'static'`. New in version 0.5.

init_jinja_globals()

Deprecated. Used to initialize the Jinja2 globals. New in version 0.5.Changed in version 0.7: This method is deprecated with 0.7. Override `create_jinja_environment()` instead.

`inject_url_defaults(endpoint, values)`

Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building. New in version 0.7.

`jinja_env`

The Jinja2 environment used to load templates.

`jinja_loader`

The Jinja loader for this package bound object. New in version 0.5.

`jinja_options = ImmutableDict({'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.with_']})`

Options that are passed directly to the Jinja2 environment.

`logger`

A `logging.Logger` object for this application. The default configuration is to log to stderr if the application is in debug mode. This logger can be used to (surprise) log messages. Here some examples:

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

New in version 0.3.

`logger_name`

The name of the logger to use. By default the logger name is the package name passed to the constructor. New in version 0.4.

`make_default_options_response()`

This method is called to create the default *OPTIONS* response. This can be changed through subclassing to change the default behaviour of *OPTIONS* responses. New in version 0.7.

`make_null_session()`

Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the `session_interface`. New in version 0.7.

`make_response(rv)`

Converts the return value from a view function to a real response object that is an instance of `response_class`.

The following types are allowed for *rv*:

response_class	the object is returned unchanged
str	a response object is created with the string as body
unicode	a response object is created with the string encoded to utf-8 as body
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

`open_resource(resource)`

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

```
/myapplication.py
/schema.sql
/static
  /style.css
/templates
  /layout.html
  /index.html
```

If you want to open the *schema.sql* file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.

`open_session(request)`

Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the `secret_key` is set. Instead of overriding this method we recommend replacing the `session_interface`.

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

`permanent_session_lifetime`

A `timedelta` which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the `PERMANENT_SESSION_LIFETIME` configuration key. Defaults to `timedelta(days=31)`

`preprocess_request()`

Called before the actual request dispatching and will call every `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.

This also triggers the `url_value_processor()` functions before the actually `before_request()` functions are called.

`preserve_context_on_exception`

Returns the value of the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration value in case it's set, otherwise a sensible default is returned. New in version 0.7.

`process_response(response)`

Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the `after_request()` decorated functions. Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

Parameters `response` – a `response_class` object.

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

`propagate_exceptions`

Returns the value of the `PROPAGATE_EXCEPTIONS` configuration value in case it's set, otherwise a sensible default is returned. New in version 0.7.

`register_blueprint(*args, **kwargs)`

Registers a blueprint on the application. New in version 0.7.

`register_error_handler(code_or_exception, f)`

Alternative error attach function to the `errorhandler()` decorator that is more straightforward to use for non decorator usage. New in version 0.7.

`register_module(module, **options)`

Registers a module with this application. The keyword argument of this function are the same as the ones for the constructor of the `Module` class and will override the values of the module if provided. Changed in version 0.7: The module system was deprecated in favor for the blueprint system.

`request_class`

The class that is used for request objects. See `Request` for more information.

alias of `Request`

`request_context(environ)`

Creates a `RequestContext` from the given environment and binds it to the current context. This must be used in combination with the *with* statement because the request is only bound to the current context for the duration of the *with* block.

Example usage:

```
with app.request_context(environ):
    do_something_with(request)
```

The object returned can also be used without the *with* statement which is useful for working in the shell. The example above is doing exactly the same as this code:

```
ctx = app.request_context(envIRON)
ctx.push()
try:
    do_something_with(request)
finally:
    ctx.pop()
```

Changed in version 0.3: Added support for non-with statement usage and *with* statement is now passed the ctx object.

Parameters *environ* – a WSGI environment

response_class

The class that is used for response objects. See Response for more information.

alias of Response

*route(rule, **options)*

A decorator that is used to register a view function for a given URL rule. Example:

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

Variable parts in the route can be specified with angular brackets (/user/<username>). By default a variable part in the URL accepts any string without a slash however a different converter can be specified as well by using <converter:name>.

Variable parts are passed to the view function as keyword arguments.

The following converters are possible:

<i>int</i>	accepts integers
<i>float</i>	like <i>int</i> but for floating point values
<i>path</i>	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*). Starting with Flask 0.6, *OPTIONS* is implicitly added and handled by the standard request handling.
- **subdomain** – specifies the rule for the subdomain 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='127.0.0.1', 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.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evaluate=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke `run()` with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

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.serving.run_simple()` for more information.

`save_session(session, response)`

Saves the session if it needs updates. For the default implementation, check `open_session()`. Instead of overriding this method we recommend replacing the `session_interface`.

Parameters

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

`secret_key`

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the `SECRET_KEY` configuration key. Defaults to *None*.

`select_jinja_autoescape(filename)`

Returns *True* if autoescaping should be active for the given template name. New in version 0.5.

`send_static_file(filename)`

Function used internally to send static files from the static folder to the browser. New in version 0.5.

`session_cookie_name`

The secure cookie uses this for the name of the session cookie.

This attribute can also be configured from the config with the `SESSION_COOKIE_NAME` configuration key. Defaults to 'session'

`session_interface = <flask.sessions.SecureCookieSessionInterface object at 0x104f965d0>`
the session interface to use. By default an instance of `SecureCookieSessionInterface` is used here. New in version 0.7.

`teardown_request(*args, **kwargs)`

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

```
ctx = app.test_request_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by `try/except` statements and log occurring errors.

teardown_request_funcs = None

A dictionary with lists of functions that are called after each request, even if an exception has occurred. The key of the dictionary is the name of the blueprint this function is active for, *None* for all requests. These functions are not allowed to modify the request, and their return values are ignored. If an exception occurred while processing the request, it gets passed to each `teardown_request` function. To register a function here, use the `teardown_request()` decorator. New in version 0.7.

template_context_processors = None

A dictionary with list of functions that are called without argument to populate the template context. The key of the dictionary is the name of the blueprint this function is active for, *None* for all requests. Each returns a dictionary that the template context is updated with. To register a function here, use the `context_processor()` decorator.

template_filter(*args, **kwargs)

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

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

Parameters name – the optional name of the filter, otherwise the function name will be used.

test_client(use_cookies=True)

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

The test client can be used in a *with* block to defer the closing down of the context until the end of the *with* block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
```

```
assert request.args['vodka'] == '42'
```

Changed in version 0.4: added support for *with* block usage for the client. New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the *test_client_class* attribute.

test_client_class = None

the test client that is used with when *test_client* is used. New in version 0.7.

test_request_context(*args, **kwargs)

Creates a WSGI environment from the given values (see `werkzeug.test.EnvironBuilder()` for more information, this function accepts the same arguments).

testing

The testing flag. Set this to *True* to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate unittest helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and `PROPAGATE_EXCEPTIONS` is not changed from the default it's implicitly enabled.

This attribute can also be configured from the config with the *TESTING* configuration key. Defaults to *False*.

trap_http_exception(e)

Checks if an HTTP exception should be trapped or not. By default this will return *False* for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to *True*. It also returns *True* if `TRAP_HTTP_EXCEPTIONS` is set to *True*.

This is called for all HTTP exceptions raised by a view function. If it returns *True* for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions. New in version 0.8.

update_template_context(context)

Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

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

url_default_functions = None

A dictionary with lists of functions that can be used as URL value preprocessors. The key *None* here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance

to modify the dictionary of URL values before they are used as the keyword arguments of the view function. For each function registered this one should also provide a `url_defaults()` function that adds the parameters automatically again that were removed that way. New in version 0.7.

`url_defaults(*args, **kwargs)`

Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

`url_map = None`

The `Map` for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

```
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def to_python(self, value):
        return value.split(',')
    def to_url(self, values):
        return ','.join(BaseConverter.to_url(value)
                        for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter
```

`url_rule_class`

The rule object to use for URL rules created. This is used by `add_url_rule()`. Defaults to `werkzeug.routing.Rule`. New in version 0.7. alias of `Rule`

`url_value_preprocessor(*args, **kwargs)`

Registers a function as URL value preprocessor for all view functions of the application. It's called before the view functions are called and can modify the url values provided.

`url_value_preprocessors = None`

A dictionary with lists of functions that can be used as URL value processor functions. Whenever a URL is built these functions are called to modify the dictionary of values in place. The key *None* here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary New in version 0.7.

`use_x_sendfile`

Enable this if you want to use the X-Sendfile feature. Keep in mind that the server has to support this. This only affects files sent with the `send_file()` method. New in version 0.2. This attribute can also be configured from the config with the `USE_X_SENDFILE` configuration key. Defaults to *False*.

`view_functions = None`

A dictionary of all view functions registered. The keys will be function names which are also used to generate URLs and the values are the function objects themselves. To register a view function, use the `route()` decorator.

`wsgi_app(environ, start_response)`

The actual WSGI application. This is not implemented in `__call__` so that middlewares can be applied without losing a reference to the class. So instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it. Changed in version 0.7: The behavior of the before and after request callbacks was changed under error conditions and a new callback was added that will always execute at the end of the request, independent on if an error occurred or not. See *Callbacks and Errors*.

Parameters

- **environ** – a WSGI environment
- **start_response** – a callable accepting a status code, a list of headers and an optional exception context to start the response

17.2 Blueprint Objects

```
class flask.Blueprint(name, import_name, static_folder=None,
                      static_url_path=None, template_folder=None,
                      url_prefix=None, subdomain=None, url_defaults=None)
```

Represents a blueprint. A blueprint is an object that records functions that will be called with the `BlueprintSetupState` later to register functions or other things on the main application. See *Modular Applications with Blueprints* for more information. New in version 0.7.

`add_url_rule(rule, endpoint=None, view_func=None, **options)`

Like `Flask.add_url_rule()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

`after_app_request(f)`

Like `Flask.after_request()` but for a blueprint. Such a function is executed after each request, even if outside of the blueprint.

`after_request(f)`

Like `Flask.after_request()` but for a blueprint. This function is only executed after each request that is handled by a function of that blueprint.

`app_context_processor(f)`

Like `Flask.context_processor()` but for a blueprint. Such a function is executed each request, even if outside of the blueprint.

`app_errorhandler(code)`
 Like `Flask.errorhandler()` but for a blueprint. This handler is used for all requests, even if outside of the blueprint.

`app_url_defaults(f)`
 Same as `url_defaults()` but application wide.

`app_url_value_preprocessor(f)`
 Same as `url_value_preprocessor()` but application wide.

`before_app_first_request(f)`
 Like `Flask.before_first_request()`. Such a function is executed before the first request to the application.

`before_app_request(f)`
 Like `Flask.before_request()`. Such a function is executed before each request, even if outside of a blueprint.

`before_request(f)`
 Like `Flask.before_request()` but for a blueprint. This function is only executed before each request that is handled by a function of that blueprint.

`context_processor(f)`
 Like `Flask.context_processor()` but for a blueprint. This function is only executed for requests handled by a blueprint.

`endpoint(endpoint)`
 Like `Flask.endpoint()` but for a blueprint. This does not prefix the endpoint with the blueprint name, this has to be done explicitly by the user of this method. If the endpoint is prefixed with a `.` it will be registered to the current blueprint, otherwise it's an application independent endpoint.

`errorhandler(code_or_exception)`
 Registers an error handler that becomes active for this blueprint only. Please be aware that routing does not happen local to a blueprint so an error handler for 404 usually is not handled by a blueprint unless it is caused inside a view function. Another special case is the 500 internal server error which is always looked up from the application.

Otherwise works as the `errorhandler()` decorator of the Flask object.

`has_static_folder`
 This is `True` if the package bound object's container has a folder named `'static'`. New in version 0.5.

`jinja_loader`
 The Jinja loader for this package bound object. New in version 0.5.

`make_setup_state(app, options, first_registration=False)`
 Creates an instance of `BlueprintSetupState()` object that is later passed to the register callback functions. Subclasses can override this to return a subclass of the setup state.

`open_resource(resource)`

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

```
/myapplication.py
/schema.sql
/static
  /style.css
/templates
  /layout.html
  /index.html
```

If you want to open the *schema.sql* file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters resource – the name of the resource. To access resources within subfolders use forward slashes as separator.

`record(func)`

Registers a function that is called when the blueprint is registered on the application. This function is called with the state as argument as returned by the `make_setup_state()` method.

`record_once(func)`

Works like `record()` but wraps the function in another function that will ensure the function is only called once. If the blueprint is registered a second time on the application, the function passed is not called.

`register(app, options, first_registration=False)`

Called by `Flask.register_blueprint()` to register a blueprint on the application. This can be overridden to customize the register behavior. Keyword arguments from `register_blueprint()` are directly forwarded to this method in the *options* dictionary.

`route(rule, **options)`

Like `Flask.route()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

`send_static_file(filename)`

Function used internally to send static files from the static folder to the browser. New in version 0.5.

`teardown_app_request(f)`

Like `Flask.teardown_request()` but for a blueprint. Such a function is executed when tearing down each request, even if outside of the blueprint.

`teardown_request(f)`

Like `Flask.teardown_request()` but for a blueprint. This function is only executed when tearing down requests handled by a function of that

blueprint. Teardown request functions are executed when the request context is popped, even when no actual request was performed.

`url_defaults(f)`

Callback function for URL defaults for this blueprint. It's called with the endpoint and values and should update the values passed in place.

`url_value_preprocessor(f)`

Registers a function as URL value preprocessor for this blueprint. It's called before the view functions are called and can modify the url values provided.

17.3 Incoming Request Data

class `flask.Request(environ, populate_request=True, shallow=False)`

The request object used by default in Flask. Remembers the matched endpoint and view arguments.

It is what ends up as `request`. If you want to replace the request object used you can subclass this and set `request_class` to your subclass.

The request object is a `Request` subclass and provides all of the attributes Werkzeug defines plus a few Flask specific ones.

`form`

A `MultDict` 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 `MultDict` with the parsed contents of the query string. (The part in the URL after the question mark).

`values`

A `CombinedMultDict` with the contents of both `form` and `args`.

`cookies`

A `dict` with the contents of all cookies transmitted with the request.

`stream`

If the incoming form data was not encoded with a known mimetype the data is stored unmodified in this stream for consumption. Most of the time it is a better idea to use `data` which will give you that data as a string. The stream only returns the data once.

`headers`

The incoming request headers as a dictionary like object.

`data`

Contains the incoming request data as string in case it came with a mimetype Flask does not handle.

files

A `MultiDict` with files uploaded as part of a *POST* or *PUT* request. Each file is stored as `FileStorage` object. It basically behaves like a standard file object you know from Python, with the difference that it also has a `save()` function that can store the file on the filesystem.

environ

The underlying WSGI environment.

method

The current request method (POST, GET etc.)

path

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:

<i>path</i>	/page.html
<i>script_root</i>	/myapplication
<i>base_url</i>	http://www.example.com/myapplication/page.html
<i>url</i>	http://www.example.com/myapplication/page.html?x=y
<i>url_root</i>	http://www.example.com/myapplication/

is_xhr

True if the request was triggered via a JavaScript *XMLHttpRequest*. This only works with libraries that support the X-Requested-With header and set it to *XMLHttpRequest*. Libraries that do that are prototype, jQuery and Mochikit and probably some more.

json

Contains the parsed body of the JSON request if the mimetype of the incoming data was *application/json*. This requires Python 2.6 or an installed version of simplejson.

blueprint

The name of the current blueprint

endpoint

The endpoint that matched the request. This in combination with `view_args`

can be used to reconstruct the same or a modified URL. If an exception happened when matching, this will be *None*.

`json`

If the mimetype is *application/json* this will contain the parsed JSON data.

`max_content_length`

Read-only view of the *MAX_CONTENT_LENGTH* config key.

`module`

The name of the current module if the request was dispatched to an actual module. This is deprecated functionality, use blueprints instead.

`on_json_loading_failed(e)`

Called if decoding of the JSON data failed. The return value of this method is used by `json` when an error occurred. The default implementation raises a `BadRequest`. New in version 0.8.

`routing_exception = None`

if matching the URL failed, this is the exception that will be raised / was raised as part of the request handling. This is usually a `NotFound` exception or something similar.

`url_rule = None`

the internal URL rule that matched the request. This can be useful to inspect which methods are allowed for the URL from a before/after handler (`request.url_rule.methods`) etc. New in version 0.6.

`view_args = None`

a dict of view arguments that matched the request. If an exception happened when matching, this will be *None*.

class `flask.request`

To access incoming request data, you can use the global *request* object. Flask parses incoming request data for you and gives you access to it through that global object. Internally Flask makes sure that you always get the correct data for the active thread if you are in a multithreaded environment.

This is a proxy. See *Notes On Proxies* for more information.

The request object is an instance of a `Request` subclass and provides all of the attributes Werkzeug defines. This just shows a quick overview of the most important ones.

17.4 Response Objects

class `flask.Response(response=None, status=None, headers=None, mimetype=None, content_type=None, direct_passthrough=False)`

The response object that is used by default in Flask. Works like the response object from Werkzeug but is set to have an HTML mimetype by default. Quite

often you don't have to create this object yourself because `make_response()` will take care of that for you.

If you want to replace the response object used you can subclass this and set `response_class` to your subclass.

`headers`

A `Headers` object representing the response headers.

`status_code`

The response status as integer.

`set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure=None, 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.
- **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.
- **path** – limits the cookie to a given path, per default it will span the whole domain.

`data`

The string representation of the request body. Whenever you access this property the request iterable is encoded and flattened. This can lead to unwanted behavior if you stream big data.

This behavior can be disabled by setting `implicit_sequence_conversion` to *False*.

`mimetype`

The mimetype (content type without charset etc.)

17.5 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 an-

other. The way Flask does this is by using a signed cookie. So the user can look at the session contents, but not modify it unless they know the secret key, so make sure to set that to something complex and unguessable.

To access the current session you can use the session object:

class flask.session

The session object works pretty much like an ordinary dict, with the difference that it keeps track on modifications.

This is a proxy. See *Notes On Proxies* for more information.

The following attributes are interesting:

new

True if the session is new, *False* otherwise.

modified

True if the session object detected a modification. Be advised that modifications on mutable structures are not picked up automatically, in that situation you have to explicitly set the attribute to *True* yourself. Here an example:

```
# this change is not picked up because a mutable object (here
# a list) is changed.
session['objects'].append(42)
# so mark it as modified yourself
session.modified = True
```

permanent

If set to *True* the session lives for permanent_session_lifetime seconds. The default is 31 days. If set to *False* (which is the default) the session will be deleted when the user closes the browser.

17.6 Session Interface

New in version 0.8. The session interface provides a simple way to replace the session implementation that Flask is using.

class flask.sessions.SessionInterface

The basic interface you have to implement in order to replace the default session interface which uses werkzeug's securecookie implementation. The only methods you have to implement are `open_session()` and `save_session()`, the others have useful defaults which you don't need to change.

The session object returned by the `open_session()` method has to provide a dictionary like interface plus the properties and methods from the `SessionMixin`. We recommend just subclassing a dict and adding that mixin:

```
class Session(dict, SessionMixin):
    pass
```

If `open_session()` returns *None* Flask will call into `make_null_session()` to create a session that acts as replacement if the session support cannot work because some requirement is not fulfilled. The default `NullSession` class that is created will complain that the secret key was not set.

To replace the session interface on an application all you have to do is to assign `flask.Flask.session_interface`:

```
app = Flask(__name__)
app.session_interface = MySessionInterface()
```

New in version 0.8.

`get_cookie_domain(app)`

Helpful helper method that returns the cookie domain that should be used for the session cookie if session cookies are used.

`get_expiration_time(app, session)`

A helper method that returns an expiration date for the session or *None* if the session is linked to the browser session. The default implementation returns now + the permanent session lifetime configured on the application.

`is_null_session(obj)`

Checks if a given object is a null session. Null sessions are not asked to be saved.

This checks if the object is an instance of `null_session_class` by default.

`make_null_session(app)`

Creates a null session which acts as a replacement object if the real session support could not be loaded due to a configuration error. This mainly aids the user experience because the job of the null session is to still support lookup without complaining but modifications are answered with a helpful error message of what failed.

This creates an instance of `null_session_class` by default.

`null_session_class`

`make_null_session()` will look here for the class that should be created when a null session is requested. Likewise the `is_null_session()` method will perform a typecheck against this type.

alias of `NullSession`

`open_session(app, request)`

This method has to be implemented and must either return *None* in case the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on `SessionMixin`.

`save_session(app, session, response)`

This is called for actual sessions returned by `open_session()` at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.

class flask.sessions.SecureCookieSessionInterface
 The cookie session interface that uses the Werkzeug securecookie as client side session backend.

session_class
 alias of SecureCookieSession

class flask.sessions.NullSession(*data=None, secret_key=None, new=True*)
 Class used to generate nicer error messages if sessions are not available. Will still allow read-only access to the empty session but fail on setting.

class flask.sessions.SessionMixin
 Expands a basic dictionary with an accessors that are expected by Flask extensions and users for the session.

modified = **True**
 for some backends this will always be *True*, but some backends will default this to false and detect changes in the dictionary for as long as changes do not happen on mutable structures in the session. The default mixin implementation just hardcodes *True* in.

new = **False**
 some session backends can tell you if a session is new, but that is not necessarily guaranteed. Use with caution. The default mixin implementation just hardcodes *False* in.

permanent
 this reflects the '_permanent' key in the dict.

17.7 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.

This is a proxy. See *Notes On Proxies* for more information.

17.8 Useful Functions and Classes

flask.current_app
 Points to the application handling the request. This is useful for extensions that want to support multiple applications running side by side.

This is a proxy. See *Notes On Proxies* for more information.

`flask.has_request_context()`

If you have code that wants to test if a request context is there or not this function can be used. For instance if you want to take advantage of request information is it's available but fail silently if the request object is unavailable.

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and has_request_context():
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

Alternatively you can also just test any of the context bound objects (such as request or g for truthness):

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and request:
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

New in version 0.7.

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

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

Variable arguments that are unknown to the target endpoint are appended to the generated URL as query arguments. If the value of a query argument is *None*, the whole pair is skipped. In case blueprints are active you can shortcut references to the same blueprint by prefixing the local endpoint with a dot (.).

This will reference the index function local to the current blueprint:

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

For more information, head over to the *Quickstart*.

Parameters

- **endpoint** – the endpoint of the URL (name of the function)
- **values** – the variable arguments of the URL rule
- **_external** – if set to *True*, an absolute URL is generated.

`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)`

Return 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.

Parameters

- **location** – the location the response should redirect to.
- **code** – the redirect status code. defaults to 302.

`flask.make_response(*args)`

Sometimes it is necessary to set additional headers in a view. Because views do not have to return response objects but can return a value that is converted into a response object by Flask itself, it becomes tricky to add headers to it. This function can be called instead of using a return and you will get a response object which you can use to attach headers.

If view looked like this and you want to add a new header:

```
def index():
    return render_template('index.html', foo=42)
```

You can now do something like this:

```
def index():
    response = make_response(render_template('index.html', foo=42))
    response.headers['X-Parachutes'] = 'parachutes are cool'
    return response
```

This function accepts the very same arguments you can return from a view function. This for example creates a response with a 404 error code:

```
response = make_response(render_template('not_found.html'), 404)
```

Internally this function does the following things:

- if no arguments are passed, it creates a new response argument
- if one argument is passed, `flask.Flask.make_response()` is invoked with it.
- if more than one argument is passed, the arguments are passed to the `flask.Flask.make_response()` function as tuple.

New in version 0.6.

`flask.send_file(filename_or_fp, mimetype=None, as_attachment=False, attachment_filename=None, add_etags=True, cache_timeout=43200, conditional=False)`

Sends the contents of a file to the client. This will use the most efficient method available and configured. By default it will try to use the WSGI

server's `file_wrapper` support. Alternatively you can set the application's `use_x_sendfile` attribute to `True` to directly emit an *X-Sendfile* header. This however requires support of the underlying webserver for *X-Sendfile*.

By default it will try to guess the `mimetype` for you, but you can also explicitly provide one. For extra security you probably want to send certain files as attachment (HTML for instance). The `mimetype` guessing requires a `filename` or an `attachment_filename` to be provided.

Please never pass filenames to this function from user sources without checking them first. Something like this is usually sufficient to avoid security problems:

```
if '..' in filename or filename.startswith('/'):
    abort(404)
```

New in version 0.2. New in version 0.5: The `add_etags`, `cache_timeout` and `conditional` parameters were added. The default behaviour is now to attach etags. Changed in version 0.7: `mimetype` guessing and etag support for file objects was deprecated because it was unreliable. Pass a filename if you are able to, otherwise attach an etag yourself. This functionality will be removed in Flask 1.0

Parameters

- **filename_or_fp** – the filename of the file to send. This is relative to the `root_path` if a relative path is specified. Alternatively a file object might be provided in which case *X-Sendfile* might not work and fall back to the traditional method. Make sure that the file pointer is positioned at the start of data to send before calling `send_file()`.
- **mimetype** – the `mimetype` of the file if provided, otherwise auto detection happens.
- **as_attachment** – set to `True` if you want to send this file with a `Content-Disposition: attachment` header.
- **attachment_filename** – the filename for the attachment if it differs from the file's filename.
- **add_etags** – set to `False` to disable attaching of etags.
- **conditional** – set to `True` to enable conditional responses.
- **cache_timeout** – the timeout in seconds for the headers.

`flask.send_from_directory(directory, filename, **options)`

Send a file from a given directory with `send_file()`. This is a secure way to quickly expose static files from an upload folder or something similar.

Example usage:

```
@app.route('/uploads/<path:filename>')
def download_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                              filename, as_attachment=True)
```

Sending files and Performance

It is strongly recommended to activate either *X-Sendfile* support in your webserver or (if no authentication happens) to tell the webserver to serve files for the given path on its own without calling into the web application for improved performance.

New in version 0.5.

Parameters

- **directory** – the directory where all the files are stored.
- **filename** – the filename relative to that directory to download.
- **options** – optional keyword arguments that are directly forwarded to `send_file()`.

`flask.safe_join(directory, filename)`
Safely join *directory* and *filename*.

Example usage:

```
@app.route('/wiki/<path:filename>')
def wiki_page(filename):
    filename = safe_join(app.config['WIKI_FOLDER'], filename)
    with open(filename, 'rb') as fd:
        content = fd.read() # Read and process the file content...
```

Parameters

- **directory** – the base directory.
- **filename** – the untrusted filename relative to that directory.

Raises `NotFound` if the resulting path would fall out of *directory*.

`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.

`unescape()`

Unescape markup again into an unicode string. This also resolves known HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

`striptags()`

Unescape markup into an unicode 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'
```

17.9 Message Flashing

`flask.flash(message, category='message')`

Flashes a message to the next request. In order to remove the flashed message from the session and to display it to the user, the template has to call `get_flashed_messages()`.

Parameters

- **message** – the message to be flashed.
- **category** – the category for the message. The following values are recommended: 'message' for any kind of message, 'error' for errors, 'info' for information messages and 'warning' for warnings. However any kind of string can be used as category.

`flask.get_flashed_messages(with_categories=False)`

Pulls all flashed messages from the session and returns them. Further calls in the same request to the function will return the same messages. By default just the messages are returned, but when *with_categories* is set to *True*, the return value will be a list of tuples in the form (category, message) instead.

Example usage:

```
{% for category, msg in get_flashed_messages(with_categories=true) %}
  <p class=flash-{{ category }}>{{ msg }}
{% endfor %}
```

Changed in version 0.3: *with_categories* parameter added.

Parameters *with_categories* – set to *True* to also receive categories.

17.10 Returning JSON

`flask.jsonify(*args, **kwargs)`

Creates a Response with the JSON representation of the given arguments with an *application/json* mimetype. The arguments to this function are the same as to the `dict` constructor.

Example usage:

```
@app.route('/_get_current_user')
def get_current_user():
    return jsonify(username=g.user.username,
                  email=g.user.email,
                  id=g.user.id)
```

This will send a JSON response like this to the browser:

```
{
  "username": "admin",
```

```

    "email": "admin@localhost",
    "id": 42
}

```

This requires Python 2.6 or an installed version of `simplejson`. For security reasons only objects are supported toplevel. For more information about this, have a look at *JSON Security*. New in version 0.2.

`flask.json`

If JSON support is picked up, this will be the module that Flask is using to parse and serialize JSON. So instead of doing this yourself:

```

try:
    import simplejson as json
except ImportError:
    import json

```

You can instead just do this:

```

from flask import json

```

For usage examples, read the `json` documentation.

The `dumps()` function of this `json` module is also available as filter called `|tojson` in Jinja2. Note that inside *script* tags no escaping must take place, so make sure to disable escaping with `|safe` if you intend to use it inside *script* tags:

```

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

```

Note that the `|tojson` filter escapes forward slashes properly.

17.11 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.

`flask.get_template_attribute(template_name, attribute)`

Loads a macro (or variable) a template exports. This can be used to invoke a macro from within Python code. If you for example have a template named `_cider.html` with the following contents:

```
{% macro hello(name) %}Hello {{ name }}!{% endmacro %}
```

You can access this from Python code like this:

```
hello = get_template_attribute('_cider.html', 'hello')
return hello('World')
```

New in version 0.2.

Parameters

- **template_name** – the name of the template
- **attribute** – the name of the variable of macro to access

17.12 Configuration

class `flask.Config(root_path, defaults=None)`

Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

Either you can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Or alternatively you can define the configuration options in the module that calls `from_object()` or provide an import path to a module that should be loaded. It is also possible to tell it to use the same module and with that provide the configuration values just before the call:

```
DEBUG = True
SECRET_KEY = 'development key'
app.config.from_object(__name__)
```

In both cases (loading from any Python file or loading from modules), only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

```
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS X use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use *set* instead.

Parameters

- **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application's root_path.
- **defaults** – an optional dictionary of default values

`from_envvar(variable_name, silent=False)`

Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

Parameters

- **variable_name** – name of the environment variable
- **silent** – set to *True* if you want silent failure for missing files.

Returns bool. *True* if able to load config, *False* otherwise.

`from_object(obj)`

Updates the values from the given object. An object can be of one of the following two types:

- a string: in this case the object with that name will be imported
- an actual object reference: that object is used directly

Objects are usually either modules or classes.

Just the uppercase variables in that object are stored in the config. Example usage:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with `from_pyfile()` and ideally from a location not within the package because the package might be installed system wide.

Parameters **obj** – an import name or object

`from_pyfile(filename, silent=False)`

Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the `from_object()` function.

Parameters

- **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to *True* if you want silent failure for missing files.

New in version 0.7: *silent* parameter.

17.13 Useful Internals

class `flask.ctx.RequestContext(app, environ)`

The request context contains all request relevant information. It is created at the beginning of the request and pushed to the `_request_ctx_stack` and removed at the end of it. It will create the URL adapter and request object for the WSGI environment provided.

Do not attempt to use this class directly, instead use `test_request_context()` and `request_context()` to create this object.

When the request context is popped, it will evaluate all the functions registered on the application for teardown execution (`teardown_request()`).

The request context is automatically popped at the end of the request for you. In debug mode the request context is kept around if exceptions happen so that interactive debuggers have a chance to introspect the data. With 0.4 this can also be forced for requests that did not fail and outside of *DEBUG* mode. By setting `'flask._preserve_context'` to *True* on the WSGI environment the context will not pop itself at the end of the request. This is used by the `test_client()` for example to implement the deferred cleanup functionality.

You might find this helpful for unittests where you need the information from the context local around for a little longer. Make sure to properly `pop()` the stack yourself in that situation, otherwise your unittests will leak memory.

`match_request()`

Can be overridden by a subclass to hook into the matching of the request.

`pop()`

Pops the request context and unbinds it by doing that. This will also trigger the execution of functions registered by the `teardown_request()` decorator.

`push()`

Binds the request context to the current context.

`flask._request_ctx_stack`

The internal `LocalStack` that is used to implement all the context local objects

used in Flask. This is a documented instance and can be used by extensions and application code but the use is discouraged in general.

The following attributes are always present on each layer of the stack:

app the active Flask application.

url_adapter the URL adapter that was used to match the request.

request the current request object.

session the active session object.

g an object with all the attributes of the flask.g object.

flashes an internal cache for the flashed messages.

Example usage:

```
from flask import _request_ctx_stack

def get_session():
    ctx = _request_ctx_stack.top
    if ctx is not None:
        return ctx.session
```

class flask.blueprints.BlueprintSetupState(*blueprint*, *app*, *options*,
first_registration)

Temporary holder object for registering a blueprint with the application. An instance of this class is created by the `make_setup_state()` method and later passed to all register callback functions.

add_url_rule(*rule*, *endpoint=None*, *view_func=None*, ***options*)

A helper method to register a rule (and optionally a view function) to the application. The endpoint is automatically prefixed with the blueprint's name.

app = None

a reference to the current application

blueprint = None

a reference to the blueprint that created this setup state.

first_registration = None

as blueprints can be registered multiple times with the application and not everything wants to be registered multiple times on it, this attribute can be used to figure out if the blueprint was registered in the past already.

options = None

a dictionary with all options that were passed to the `register_blueprint()` method.

subdomain = None

The subdomain that the blueprint should be active for, *None* otherwise.

`url_defaults = None`

A dictionary with URL defaults that is added to each and every URL that was defined with the blueprint.

`url_prefix = None`

The prefix that should be used for all URLs defined on the blueprint.

17.14 Signals

New in version 0.6.

`flask.signals_available`

True if the signalling system is available. This is the case when [blinker](#) is installed.

`flask.template_rendered`

This signal is sent when a template was successfully rendered. The signal is invoked with the instance of the template as *template* and the context as dictionary (named *context*).

`flask.request_started`

This signal is sent before any request processing started but when the request context was set up. Because the request context is already bound, the subscriber can access the request with the standard global proxies such as `request`.

`flask.request_finished`

This signal is sent right before the response is sent to the client. It is passed the response to be sent named *response*.

`flask.got_request_exception`

This signal is sent when an exception happens during request processing. It is sent *before* the standard exception handling kicks in and even in debug mode, where no exception handling happens. The exception itself is passed to the subscriber as *exception*.

`flask.request_tearing_down`

This signal is sent when the application is tearing down the request. This is always called, even if an error happened. No arguments are provided.

class `flask.signals.Namespace`

An alias for [blinker.base.Namespace](#) if [blinker](#) is available, otherwise a dummy class that creates fake signals. This class is available for Flask extensions that want to provide the same fallback system as Flask itself.

`signal(name, doc=None)`

Creates a new signal for this namespace if [blinker](#) is available, otherwise returns a fake signal that has a `send` method that will do nothing but will fail with a [RuntimeError](#) for all other operations, including connecting.

17.15 Class Based Views

New in version 0.7.

class flask.views.View

Alternative way to use view functions. A subclass has to implement `dispatch_request()` which is called with the view arguments from the URL routing system. If methods is provided the methods do not have to be passed to the `add_url_rule()` method explicitly:

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

```
    def dispatch_request(self, name):
        return 'Hello %s!' % name
```

```
app.add_url_rule('/hello/<name>', view_func=MyView.as_view('myview'))
```

classmethod `as_view(name, *class_args, **class_kwargs)`

Converts the class into an actual view function that can be used with the routing system. What it does internally is generating a function on the fly that will instantiate the View on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

`dispatch_request()`

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

class flask.views.MethodView

Like a regular class based view but that dispatches requests to particular methods. For instance if you implement a method called `get()` it means you will response to 'GET' requests and the `dispatch_request()` implementation will automatically forward your request to that. Also options is set for you automatically:

```
class CounterAPI(MethodView):
```

```
    def get(self):
        return session.get('counter', 0)
```

```
    def post(self):
        session['counter'] = session.get('counter', 0) + 1
        return 'OK'
```

```
app.add_url_rule('/counter', view_func=CounterAPI.as_view('counter'))
```

17.16 View Function Options

For internal usage the view functions can have some attributes attached to customize behavior the view function would normally not have control over. The following attributes can be provided optionally to either override some defaults to `add_url_rule()` or general behavior:

- `__name__`: The name of a function is by default used as endpoint. If endpoint is provided explicitly this value is used. Additionally this will be prefixed with the name of the blueprint by default which cannot be customized from the function itself.
- `methods`: If methods are not provided when the URL rule is added, Flask will look on the view function object itself if a `methods` attribute exists. If it does, it will pull the information for the methods from there.
- `provide_automatic_options`: if this attribute is set Flask will either force enable or disable the automatic implementation of the HTTP `OPTIONS` response. This can be useful when working with decorators that want to customize the `OPTIONS` response on a per-view basis.

Full example:

```
def index():
    if request.method == 'OPTIONS':
        # custom options handling here
        ...
    return 'Hello World!'
index.provide_automatic_options = False
index.methods = ['GET', 'OPTIONS']

app.add_url_rule('/', index)
```

New in version 0.8: The `provide_automatic_options` functionality was added.

Part III

ADDITIONAL NOTES

Design notes, legal information and changelog are here for the interested.

DESIGN DECISIONS IN FLASK

If you are curious why Flask does certain things the way it does and not differently, this section is for you. This should give you an idea about some of the design decisions that may appear arbitrary and surprising at first, especially in direct comparison with other frameworks.

18.1 The Explicit Application Object

A Python web application based on WSGI has to have one central callable object that implements the actual application. In Flask this is an instance of the `Flask` class. Each Flask application has to create an instance of this class itself and pass it the name of the module, but why can't Flask do that itself?

Without such an explicit application object the following code:

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

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

Would look like this instead:

```
from hypothetical_flask import route

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

There are three major reasons for this. The most important one is that implicit application objects require that there may only be one instance at the time. There are ways to fake multiple applications with a single application object, like maintaining a stack of applications, but this causes some problems I won't outline here in detail. Now the question is: when does a microframework need more than one application at the same time? A good example for this is `unittesting`. When you want to test something

it can be very helpful to create a minimal application to test specific behavior. When the application object is deleted everything it allocated will be freed again.

Another thing that becomes possible when you have an explicit object lying around in your code is that you can subclass the base class (`Flask`) to alter specific behaviour. This would not be possible without hacks if the object were created ahead of time for you based on a class that is not exposed to you.

But there is another very important reason why Flask depends on an explicit instantiation of that class: the package name. Whenever you create a Flask instance you usually pass it `__name__` as package name. Flask depends on that information to properly load resources relative to your module. With Python's outstanding support for reflection it can then access the package to figure out where the templates and static files are stored (see `open_resource()`). Now obviously there are frameworks around that do not need any configuration and will still be able to load templates relative to your application module. But they have to use the current working directory for that, which is a very unreliable way to determine where the application is. The current working directory is process-wide and if you are running multiple applications in one process (which could happen in a webserver without you knowing) the paths will be off. Worse: many web-servers do not set the working directory to the directory of your application but to the document root which does not have to be the same folder.

The third reason is "explicit is better than implicit". That object is your WSGI application, you don't have to remember anything else. If you want to apply a WSGI middleware, just wrap it and you're done (though there are better ways to do that so that you do not lose the reference to the application object `wsgi_app()`).

Furthermore this design makes it possible to use a factory function to create the application which is very helpful for unittesting and similar things (*Application Factories*).

18.2 One Template Engine

Flask decides on one template engine: Jinja2. Why doesn't Flask have a pluggable template engine interface? You can obviously use a different template engine, but Flask will still configure Jinja2 for you. While that limitation that Jinja2 is *always* configured will probably go away, the decision to bundle one template engine and use that will not.

Template engines are like programming languages and each of those engines has a certain understanding about how things work. On the surface they all work the same: you tell the engine to evaluate a template with a set of variables and take the return value as string.

But that's about where similarities end. Jinja2 for example has an extensive filter system, a certain way to do template inheritance, support for reusable blocks (macros) that can be used from inside templates and also from Python code, uses Unicode for all operations, supports iterative template rendering, configurable syntax and more. On the other hand an engine like Genshi is based on XML stream evaluation, template

inheritance by taking the availability of XPath into account and more. Mako on the other hand treats templates similar to Python modules.

When it comes to connecting a template engine with an application or framework there is more than just rendering templates. For instance, Flask uses Jinja2's extensive autoescaping support. Also it provides ways to access macros from Jinja2 templates.

A template abstraction layer that would not take the unique features of the template engines away is a science on its own and a too large undertaking for a microframework like Flask.

Furthermore extensions can then easily depend on one template language being present. You can easily use your own templating language, but an extension could still depend on Jinja itself.

18.3 Micro with Dependencies

Why does Flask call itself a microframework and yet it depends on two libraries (namely Werkzeug and Jinja2). Why shouldn't it? If we look over to the Ruby side of web development there we have a protocol very similar to WSGI. Just that it's called Rack there, but besides that it looks very much like a WSGI rendition for Ruby. But nearly all applications in Ruby land do not work with Rack directly, but on top of a library with the same name. This Rack library has two equivalents in Python: WebOb (formerly Paste) and Werkzeug. Paste is still around but from my understanding it's sort of deprecated in favour of WebOb. The development of WebOb and Werkzeug started side by side with similar ideas in mind: be a good implementation of WSGI for other applications to take advantage.

Flask is a framework that takes advantage of the work already done by Werkzeug to properly interface WSGI (which can be a complex task at times). Thanks to recent developments in the Python package infrastructure, packages with dependencies are no longer an issue and there are very few reasons against having libraries that depend on others.

18.4 Thread Locals

Flask uses thread local objects (context local objects in fact, they support greenlet contexts as well) for request, session and an extra object you can put your own things on (g). Why is that and isn't that a bad idea?

Yes it is usually not such a bright idea to use thread locals. They cause troubles for servers that are not based on the concept of threads and make large applications harder to maintain. However Flask is just not designed for large applications or asynchronous servers. Flask wants to make it quick and easy to write a traditional web application.

Also see the *Becoming Big* section of the documentation for some inspiration for larger applications based on Flask.

18.5 What Flask is, What Flask is Not

Flask will never have a database layer. It will not have a form library or anything else in that direction. Flask itself just bridges to Werkzeug to implement a proper WSGI application and to Jinja2 to handle templating. It also binds to a few common standard library packages such as logging. Everything else is up for extensions.

Why is this the case? Because people have different preferences and requirements and Flask could not meet those if it would force any of this into the core. The majority of web applications will need a template engine in some sort. However not every application needs a SQL database.

The idea of Flask is to build a good foundation for all applications. Everything else is up to you or extensions.

HTML/XHTML FAQ

The Flask documentation and example applications are using HTML5. You may notice that in many situations, when end tags are optional they are not used, so that the HTML is cleaner and faster to load. Because there is much confusion about HTML and XHTML among developers, this document tries to answer some of the major questions.

19.1 History of XHTML

For a while, it appeared that HTML was about to be replaced by XHTML. However, barely any websites on the Internet are actual XHTML (which is HTML processed using XML rules). There are a couple of major reasons why this is the case. One of them is Internet Explorer's lack of proper XHTML support. The XHTML spec states that XHTML must be served with the MIME type *application/xhtml+xml*, but Internet Explorer refuses to read files with that MIME type. While it is relatively easy to configure Web servers to serve XHTML properly, few people do. This is likely because properly using XHTML can be quite painful.

One of the most important causes of pain is XML's draconian (strict and ruthless) error handling. When an XML parsing error is encountered, the browser is supposed to show the user an ugly error message, instead of attempting to recover from the error and display what it can. Most of the (X)HTML generation on the web is based on non-XML template engines (such as Jinja, the one used in Flask) which do not protect you from accidentally creating invalid XHTML. There are XML based template engines, such as Kid and the popular Genshi, but they often come with a larger runtime overhead and, are not as straightforward to use because they have to obey XML rules.

The majority of users, however, assumed they were properly using XHTML. They wrote an XHTML doctype at the top of the document and self-closed all the necessary tags (
 becomes
 or
</br> in XHTML). However, even if the document properly validates as XHTML, what really determines XHTML/HTML processing in browsers is the MIME type, which as said before is often not set properly. So the valid XHTML was being treated as invalid HTML.

XHTML also changed the way JavaScript is used. To properly work with XHTML, programmers have to use the namespaced DOM interface with the XHTML namespace

to query for HTML elements.

19.2 History of HTML5

Development of the HTML5 specification was started in 2004 under the name “Web Applications 1.0” by the Web Hypertext Application Technology Working Group, or WHATWG (which was formed by the major browser vendors Apple, Mozilla, and Opera) with the goal of writing a new and improved HTML specification, based on existing browser behaviour instead of unrealistic and backwards-incompatible specifications.

For example, in HTML4 `<title>Hello/` theoretically parses exactly the same as `<title>Hello</title>`. However, since people were using XHTML-like tags along the lines of `<link />`, browser vendors implemented the XHTML syntax over the syntax defined by the specification.

In 2007, the specification was adopted as the basis of a new HTML specification under the umbrella of the W3C, known as HTML5. Currently, it appears that XHTML is losing traction, as the XHTML 2 working group has been disbanded and HTML5 is being implemented by all major browser vendors.

19.3 HTML versus XHTML

The following table gives you a quick overview of features available in HTML 4.01, XHTML 1.1 and HTML5. (XHTML 1.0 is not included, as it was superseded by XHTML 1.1 and the barely-used XHTML5.)

	HTML4.01	XHTML1.1	HTML5
<code><tag/value/ == <tag>value</tag></code>	✓ ¹	✗	✗
<code>
</code> supported	✗	✓	✓ ²
<code><script/></code> supported	✗	✓	✗
should be served as <i>text/html</i>	✓	✗ ³	✓
should be served as <i>application/xhtml+xml</i>	✗	✓	✗
strict error handling	✗	✓	✗
inline SVG	✗	✓	✓
inline MathML	✗	✓	✓
<code><video></code> tag	✗	✗	✓
<code><audio></code> tag	✗	✗	✓
New semantic tags like <code><article></code>	✗	✗	✓

¹This is an obscure feature inherited from SGML. It is usually not supported by browsers, for reasons

19.4 What does “strict” mean?

HTML5 has strictly defined parsing rules, but it also specifies exactly how a browser should react to parsing errors - unlike XHTML, which simply states parsing should abort. Some people are confused by apparently invalid syntax that still generates the expected results (for example, missing end tags or unquoted attribute values).

Some of these work because of the lenient error handling most browsers use when they encounter a markup error, others are actually specified. The following constructs are optional in HTML5 by standard, but have to be supported by browsers:

- Wrapping the document in an `<html>` tag
- Wrapping header elements in `<head>` or the body elements in `<body>`
- Closing the `<p>`, `<li>`, `<dt>`, `<dd>`, `<tr>`, `<td>`, `<th>`, `<tbody>`, `<thead>`, or `<tfoot>` tags.
- Quoting attributes, so long as they contain no whitespace or special characters (like `<`, `>`, `'`, or `"`).
- Requiring boolean attributes to have a value.

This means the following page in HTML5 is perfectly valid:

```
<!doctype html>
<title>Hello HTML5</title>
<div class=header>
  <h1>Hello HTML5</h1>
  <p class=tagline>HTML5 is awesome
</div>
<ul class=nav>
  <li><a href=/index>Index</a>
  <li><a href=/downloads>Downloads</a>
  <li><a href=/about>About</a>
</ul>
<div class=body>
  <h2>HTML5 is probably the future</h2>
  <p>
    There might be some other things around but in terms of
    browser vendor support, HTML5 is hard to beat.
  <dl>
    <dt>Key 1
    <dd>Value 1
    <dt>Key 2
    <dd>Value 2
  </dl>
</div>
```

detailed above.

²This is for compatibility with server code that generates XHTML for tags such as `<br>`. It should not be used in new code.

³XHTML 1.0 is the last XHTML standard that allows to be served as *text/html* for backwards compatibility reasons.

19.5 New technologies in HTML5

HTML5 adds many new features that make Web applications easier to write and to use.

- The `<audio>` and `<video>` tags provide a way to embed audio and video without complicated add-ons like QuickTime or Flash.
- Semantic elements like `<article>`, `<header>`, `<nav>`, and `<time>` that make content easier to understand.
- The `<canvas>` tag, which supports a powerful drawing API, reducing the need for server-generated images to present data graphically.
- New form control types like `<input type="date">` that allow user agents to make entering and validating values easier.
- Advanced JavaScript APIs like Web Storage, Web Workers, Web Sockets, geolocation, and offline applications.

Many other features have been added, as well. A good guide to new features in HTML5 is Mark Pilgrim's soon-to-be-published book, [Dive Into HTML5](#). Not all of them are supported in browsers yet, however, so use caution.

19.6 What should be used?

Currently, the answer is HTML5. There are very few reasons to use XHTML considering the latest developments in Web browsers. To summarize the reasons given above:

- Internet Explorer (which, sadly, currently leads in market share) has poor support for XHTML.
- Many JavaScript libraries also do not support XHTML, due to the more complicated namespacing API it requires.
- HTML5 adds several new features, including semantic tags and the long-awaited `<audio>` and `<video>` tags.
- It has the support of most browser vendors behind it.
- It is much easier to write, and more compact.

For most applications, it is undoubtedly better to use HTML5 than XHTML.

SECURITY CONSIDERATIONS

Web applications usually face all kinds of security problems and it's very hard to get everything right. Flask tries to solve a few of these things for you, but there are a couple more you have to take care of yourself.

20.1 Cross-Site Scripting (XSS)

Cross site scripting is the concept of injecting arbitrary HTML (and with it JavaScript) into the context of a website. To remedy this, developers have to properly escape text so that it cannot include arbitrary HTML tags. For more information on that have a look at the Wikipedia article on [Cross-Site Scripting](#).

Flask configures Jinja2 to automatically escape all values unless explicitly told otherwise. This should rule out all XSS problems caused in templates, but there are still other places where you have to be careful:

- generating HTML without the help of Jinja2
- calling Markup on data submitted by users
- sending out HTML from uploaded files, never do that, use the *Content-Disposition: attachment* header to prevent that problem.
- sending out textfiles from uploaded files. Some browsers are using content-type guessing based on the first few bytes so users could trick a browser to execute HTML.

Another thing that is very important are unquoted attributes. While Jinja2 can protect you from XSS issues by escaping HTML, there is one thing it cannot protect you from: XSS by attribute injection. To counter this possible attack vector, be sure to always quote your attributes with either double or single quotes when using Jinja expressions in them:

```
<a href="{{ href }}">the text</a>
```

Why is this necessary? Because if you would not be doing that, an attacker could easily inject custom JavaScript handlers. For example an attacker could inject this piece of HTML+JavaScript:

```
onmouseover=alert(document.cookie)
```

When the user would then move with the mouse over the link, the cookie would be presented to the user in an alert window. But instead of showing the cookie to the user, a good attacker might also execute any other JavaScript code. In combination with CSS injections the attacker might even make the element fill out the entire page so that the user would just have to have the mouse anywhere on the page to trigger the attack.

20.2 Cross-Site Request Forgery (CSRF)

Another big problem is CSRF. This is a very complex topic and I won't outline it here in detail just mention what it is and how to theoretically prevent it.

If your authentication information is stored in cookies, you have implicit state management. The state of "being logged in" is controlled by a cookie, and that cookie is sent with each request to a page. Unfortunately that includes requests triggered by 3rd party sites. If you don't keep that in mind, some people might be able to trick your application's users with social engineering to do stupid things without them knowing.

Say you have a specific URL that, when you sent *POST* requests to will delete a user's profile (say *http://example.com/user/delete*). If an attacker now creates a page that sends a post request to that page with some JavaScript they just has to trick some users to load that page and their profiles will end up being deleted.

Imagine you were to run Facebook with millions of concurrent users and someone would send out links to images of little kittens. When users would go to that page, their profiles would get deleted while they are looking at images of fluffy cats.

How can you prevent that? Basically for each request that modifies content on the server you would have to either use a one-time token and store that in the cookie **and** also transmit it with the form data. After receiving the data on the server again, you would then have to compare the two tokens and ensure they are equal.

Why does Flask not do that for you? The ideal place for this to happen is the form validation framework, which does not exist in Flask.

20.3 JSON Security

ECMAScript 5 Changes

Starting with ECMAScript 5 the behavior of literals changed. Now they are not constructed with the constructor of Array and others, but with the builtin constructor of Array which closes this particular attack vector.

JSON itself is a high-level serialization format, so there is barely anything that could cause security problems, right? You can't declare recursive structures that could cause

problems and the only thing that could possibly break are very large responses that can cause some kind of denial of service at the receiver's side.

However there is a catch. Due to how browsers work the CSRF issue comes up with JSON unfortunately. Fortunately there is also a weird part of the JavaScript specification that can be used to solve that problem easily and Flask is kinda doing that for you by preventing you from doing dangerous stuff. Unfortunately that protection is only there for `jsonify()` so you are still at risk when using other ways to generate JSON.

So what is the issue and how to avoid it? The problem are arrays at top-level in JSON. Imagine you send the following data out in a JSON request. Say that's exporting the names and email addresses of all your friends for a part of the user interface that is written in JavaScript. Not very uncommon:

```
[
  { "username": "admin",
    "email": "admin@localhost" }
]
```

And it is doing that of course only as long as you are logged in and only for you. And it is doing that for all *GET* requests to a certain URL, say the URL for that request is `http://example.com/api/get_friends.json`.

So now what happens if a clever hacker is embedding this to his website and social engineers a victim to visiting his site:

```
<script type=text/javascript>
var captured = [];
var oldArray = Array;
function Array() {
  var obj = this, id = 0, capture = function(value) {
    obj.__defineSetter__(id++, capture);
    if (value)
      captured.push(value);
  };
  capture();
}
</script>
<script type=text/javascript
  src=http://example.com/api/get_friends.json></script>
<script type=text/javascript>
Array = oldArray;
// now we have all the data in the captured array.
</script>
```

If you know a bit of JavaScript internals you might know that it's possible to patch constructors and register callbacks for setters. An attacker can use this (like above) to get all the data you exported in your JSON file. The browser will totally ignore the `application/json` mimetype if `text/javascript` is defined as content type in the script tag and evaluate that as JavaScript. Because top-level array elements are allowed (albeit useless) and we hooked in our own constructor, after that page loaded the data from the JSON response is in the *captured* array.

Because it is a syntax error in JavaScript to have an object literal (`{...}`) toplevel an attacker could not just do a request to an external URL with the script tag to load up the data. So what Flask does is to only allow objects as toplevel elements when using `jsonify()`. Make sure to do the same when using an ordinary JSON generate function.

UNICODE IN FLASK

Flask like Jinja2 and Werkzeug is totally Unicode based when it comes to text. Not only these libraries, also the majority of web related Python libraries that deal with text. If you don't know Unicode so far, you should probably read [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets](#). This part of the documentation just tries to cover the very basics so that you have a pleasant experience with Unicode related things.

21.1 Automatic Conversion

Flask has a few assumptions about your application (which you can change of course) that give you basic and painless Unicode support:

- the encoding for text on your website is UTF-8
- internally you will always use Unicode exclusively for text except for literal strings with only ASCII character points.
- encoding and decoding happens whenever you are talking over a protocol that requires bytes to be transmitted.

So what does this mean to you?

HTTP is based on bytes. Not only the protocol, also the system used to address documents on servers (so called URIs or URLs). However HTML which is usually transmitted on top of HTTP supports a large variety of character sets and which ones are used, are transmitted in an HTTP header. To not make this too complex Flask just assumes that if you are sending Unicode out you want it to be UTF-8 encoded. Flask will do the encoding and setting of the appropriate headers for you.

The same is true if you are talking to databases with the help of SQLAlchemy or a similar ORM system. Some databases have a protocol that already transmits Unicode and if they do not, SQLAlchemy or your other ORM should take care of that.

21.2 The Golden Rule

So the rule of thumb: if you are not dealing with binary data, work with Unicode. What does working with Unicode in Python 2.x mean?

- as long as you are using ASCII charpoints only (basically numbers, some special characters of latin letters without umlauts or anything fancy) you can use regular string literals ('Hello World').
- if you need anything else than ASCII in a string you have to mark this string as Unicode string by prefixing it with a lowercase *u*. (like `u'Hänsel und Gretel'`)
- if you are using non-Unicode characters in your Python files you have to tell Python which encoding your file uses. Again, I recommend UTF-8 for this purpose. To tell the interpreter your encoding you can put the `# -*- coding: utf-8 -*-` into the first or second line of your Python source file.
- Jinja is configured to decode the template files from UTF-8. So make sure to tell your editor to save the file as UTF-8 there as well.

21.3 Encoding and Decoding Yourself

If you are talking with a filesystem or something that is not really based on Unicode you will have to ensure that you decode properly when working with Unicode interface. So for example if you want to load a file on the filesystem and embed it into a Jinja2 template you will have to decode it from the encoding of that file. Here the old problem that text files do not specify their encoding comes into play. So do yourself a favour and limit yourself to UTF-8 for text files as well.

Anyways. To load such a file with Unicode you can use the built-in `str.decode()` method:

```
def read_file(filename, charset='utf-8'):
    with open(filename, 'r') as f:
        return f.read().decode(charset)
```

To go from Unicode into a specific charset such as UTF-8 you can use the `unicode.encode()` method:

```
def write_file(filename, contents, charset='utf-8'):
    with open(filename, 'w') as f:
        f.write(contents.encode(charset))
```

21.4 Configuring Editors

Most editors save as UTF-8 by default nowadays but in case your editor is not configured to do this you have to change it. Here some common ways to set your editor to store as UTF-8:

- Vim: put `set enc=utf-8` to your `.vimrc` file.
- Emacs: either use an encoding cookie or put this into your `.emacs` file:

```
(prefer-coding-system 'utf-8)  
(setq default-buffer-file-coding-system 'utf-8)
```

- Notepad++:
 1. Go to *Settings -> Preferences ...*
 2. Select the “New Document/Default Directory” tab
 3. Select “UTF-8 without BOM” as encoding

It is also recommended to use the Unix newline format, you can select it in the same panel but this is not a requirement.

FLASK EXTENSION DEVELOPMENT

Flask, being a microframework, often requires some repetitive steps to get a third party library working. Because very often these steps could be abstracted to support multiple projects the [Flask Extension Registry](#) was created.

If you want to create your own Flask extension for something that does not exist yet, this guide to extension development will help you get your extension running in no time and to feel like users would expect your extension to behave.

22.1 Anatomy of an Extension

Extensions are all located in a package called `flaskext.something` where “something” is the name of the library you want to bridge. So for example if you plan to add support for a library named *simplexml* to Flask, you would name your extension’s package `flaskext.simplexml`.

The name of the actual extension (the human readable name) however would be something like “Flask-SimpleXML”. Make sure to include the name “Flask” somewhere in that name and that you check the capitalization. This is how users can then register dependencies to your extension in their *setup.py* files.

The magic that makes it possible to have your library in a package called `flaskext.something` is called a “namespace package”. Check out the guide below how to create something like that.

But how do extensions look like themselves? An extension has to ensure that it works with multiple Flask application instances at once. This is a requirement because many people will use patterns like the *Application Factories* pattern to create their application as needed to aid unittests and to support multiple configurations. Because of that it is crucial that your application supports that kind of behaviour.

Most importantly the extension must be shipped with a *setup.py* file and registered on PyPI. Also the development checkout link should work so that people can easily install the development version into their virtualenv without having to download the library by hand.

Flask extensions must be licensed as BSD or MIT or a more liberal license to be enlisted on the Flask Extension Registry. Keep in mind that the Flask Extension Registry is a

moderated place and libraries will be reviewed upfront if they behave as required.

22.2 “Hello Flaskext!”

So let’s get started with creating such a Flask extension. The extension we want to create here will provide very basic support for SQLite3.

There is a script on github called [Flask Extension Wizard](#) which helps you create the initial folder structure. But for this very basic example we want to create all by hand to get a better feeling for it.

First we create the following folder structure:

```
flask-sqlite3/  
  flaskext/  
    __init__.py  
    sqlite3.py  
  setup.py  
  LICENSE
```

Here’s the contents of the most important files:

22.2.1 flaskext/__init__.py

The only purpose of this file is to mark the package as namespace package. This is required so that multiple modules from different PyPI packages can reside in the same Python package:

```
__import__('pkg_resources').declare_namespace(__name__)
```

If you want to know exactly what is happening there, checkout the distribute or setuptools docs which explain how this works.

Just make sure to not put anything else in there!

22.2.2 setup.py

The next file that is absolutely required is the *setup.py* file which is used to install your Flask extension. The following contents are something you can work with:

```
"""  
Flask-SQLite3  
-----  
  
This is the description for that library  
"""  
from setuptools import setup
```

```

setup(
    name='Flask-SQLite3',
    version='1.0',
    url='http://example.com/flask-sqlite3/',
    license='BSD',
    author='Your Name',
    author_email='your-email@example.com',
    description='Very short description',
    long_description=__doc__,
    packages=['flaskext'],
    namespace_packages=['flaskext'],
    zip_safe=False,
    platforms='any',
    install_requires=[
        'Flask'
    ],
    classifiers=[
        'Environment :: Web Environment',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        'Topic :: Software Development :: Libraries :: Python Modules'
    ]
)

```

That's a lot of code but you can really just copy/paste that from existing extensions and adapt. This is also what the wizard creates for you if you use it.

22.2.3 flaskext/sqlite3.py

Now this is where your extension code goes. But how exactly should such an extension look like? What are the best practices? Continue reading for some insight.

22.3 Initializing Extensions

Many extensions will need some kind of initialization step. For example, consider your application is currently connecting to SQLite like the documentation suggests (*Using SQLite 3 with Flask*) you will need to provide a few functions and before / after request handlers. So how does the extension know the name of the application object?

Quite simple: you pass it to it.

There are two recommended ways for an extension to initialize:

initialization functions: If your extension is called *helloworld* you might have a function called `init_helloworld(app[, extra_args])` that initializes the extension

for that application. It could attach before / after handlers etc.

classes: Classes work mostly like initialization functions but can later be used to further change the behaviour. For an example look at how the [OAuth extension](#) works: there is an *OAuth* object that provides some helper functions like *OAuth.remote_app* to create a reference to a remote application that uses OAuth.

What to use depends on what you have in mind. For the SQLite 3 extension we will use the class based approach because it will provide users with a manager object that handles opening and closing database connections.

22.4 The Extension Code

Here's the contents of the *flaskext/sqlite3.py* for copy/paste:

```
from __future__ import absolute_import
import sqlite3

from flask import _request_ctx_stack

class SQLite3(object):

    def __init__(self, app):
        self.app = app
        self.app.config.setdefault('SQLITE3_DATABASE', ':memory:')
        self.app.teardown_request(self.teardown_request)
        self.app.before_request(self.before_request)

    def connect(self):
        return sqlite3.connect(self.app.config['SQLITE3_DATABASE'])

    def before_request(self):
        ctx = _request_ctx_stack.top
        ctx.sqlite3_db = self.connect()

    def teardown_request(self, exception):
        ctx = _request_ctx_stack.top
        ctx.sqlite3_db.close()

    def get_db(self):
        ctx = _request_ctx_stack.top
        if ctx is not None:
            return ctx.sqlite3_db
```

So here's what these lines of code do:

1. The `__future__` import is necessary to activate absolute imports. Otherwise we could not call our module *sqlite3.py* and import the top-level *sqlite3* module which actually implements the connection to SQLite.

2. We create a class for our extension that requires a supplied *app* object, sets a configuration for the database if it's not there (`dict.setdefault()`), and attaches *before_request* and *teardown_request* handlers.
3. Next, we define a *connect* function that opens a database connection.
4. Then we set up the request handlers we bound to the app above. Note here that we're attaching our database connection to the top request context via *_request_ctx_stack.top*. Extensions should use the top context and not the *g* object to store things like database connections.
5. Finally, we add a *get_db* function that simplifies access to the context's database.

So why did we decide on a class based approach here? Because using our extension looks something like this:

```
from flask import Flask
from flaskext.sqlite3 import SQLite3

app = Flask(__name__)
app.config.from_pyfile('the-config.cfg')
manager = SQLite3(app)
db = manager.get_db()
```

You can then use the database from views like this:

```
@app.route('/')
def show_all():
    cur = db.cursor()
    cur.execute(...)
```

Opening a database connection from outside a view function is simple.

```
>>> from yourapplication import db
>>> cur = db.cursor()
>>> cur.execute(...)
```

22.5 Adding an *init_app* Function

In practice, you'll almost always want to permit users to initialize your extension and provide an app object after the fact. This can help avoid circular import problems when a user is breaking their app into multiple files. Our extension could add an *init_app* function as follows:

```
class SQLite3(object):

    def __init__(self, app=None):
        if app is not None:
            self.app = app
            self.init_app(self.app)
        else:
```

```

        self.app = None

    def init_app(self, app):
        self.app = app
        self.app.config.setdefault('SQLITE3_DATABASE', ':memory:')
        self.app.teardown_request(self.teardown_request)
        self.app.before_request(self.before_request)

    def connect(self):
        return sqlite3.connect(app.config['SQLITE3_DATABASE'])

    def before_request(self):
        ctx = _request_ctx_stack.top
        ctx.sqlite3_db = self.connect()

    def teardown_request(self, exception):
        ctx = _request_ctx_stack.top
        ctx.sqlite3_db.close()

    def get_db(self):
        ctx = _request_ctx_stack.top
        if ctx is not None:
            return ctx.sqlite3_db

```

The user could then initialize the extension in one file:

```
manager = SQLite3()
```

and bind their app to the extension in another file:

```
manager.init_app(app)
```

22.6 End-Of-Request Behavior

Due to the change in Flask 0.7 regarding functions that are run at the end of the request your extension will have to be extra careful there if it wants to continue to support older versions of Flask. The following pattern is a good way to support both:

```

def close_connection(response):
    ctx = _request_ctx_stack.top
    ctx.sqlite3_db.close()
    return response

if hasattr(app, 'teardown_request'):
    app.teardown_request(close_connection)
else:
    app.after_request(close_connection)

```

Strictly speaking the above code is wrong, because teardown functions are passed the exception and typically don't return anything. However because the return value is

discarded this will just work assuming that the code in between does not touch the passed parameter.

22.7 Learn from Others

This documentation only touches the bare minimum for extension development. If you want to learn more, it's a very good idea to check out existing extensions on the [Flask Extension Registry](#). If you feel lost there is still the [mailinglist](#) and the [IRC channel](#) to get some ideas for nice looking APIs. Especially if you do something nobody before you did, it might be a very good idea to get some more input. This not only to get an idea about what people might want to have from an extension, but also to avoid having multiple developers working on pretty much the same side by side.

Remember: good API design is hard, so introduce your project on the mailinglist, and let other developers give you a helping hand with designing the API.

The best Flask extensions are extensions that share common idioms for the API. And this can only work if collaboration happens early.

22.8 Approved Extensions

Flask also has the concept of approved extensions. Approved extensions are tested as part of Flask itself to ensure extensions do not break on new releases. These approved extensions are listed on the [Flask Extension Registry](#) and marked appropriately. If you want your own extension to be approved you have to follow these guidelines:

1. An approved Flask extension must provide exactly one package or module inside the *flaskext* namespace package.
2. It must ship a testing suite that can either be invoked with `make test` or `python setup.py test`. For test suites invoked with `make test` the extension has to ensure that all dependencies for the test are installed automatically, in case of `python setup.py test` dependencies for tests alone can be specified in the *setup.py* file. The test suite also has to be part of the distribution.
3. APIs of approved extensions will be checked for the following characteristics:
 - an approved extension has to support multiple applications running in the same Python process.
 - it must be possible to use the factory pattern for creating applications.
4. The license must be BSD/MIT/WTFPL licensed.
5. The naming scheme for official extensions is *Flask-ExtensionName* or *ExtensionName-Flask*.
6. Approved extensions must define all their dependencies in the *setup.py* file unless a dependency cannot be met because it is not available on PyPI.

7. The extension must have documentation that uses one of the two Flask themes for Sphinx documentation.
8. The setup.py description (and thus the PyPI description) has to link to the documentation, website (if there is one) and there must be a link to automatically install the development version (PackageName==dev).
9. The zip_safe flag in the setup script must be set to False, even if the extension would be safe for zipping.
10. An extension currently has to support Python 2.5, 2.6 as well as Python 2.7

POCOO STYLEGUIDE

The Pycoc styleguide is the styleguide for all Pycoc Projects, including Flask. This styleguide is a requirement for Patches to Flask and a recommendation for Flask extensions.

In general the Pycoc Styleguide closely follows [PEP 8](#) with some small differences and extensions.

23.1 General Layout

Indentation: 4 real spaces. No tabs, no exceptions.

Maximum line length: 79 characters with a soft limit for 84 if absolutely necessary. Try to avoid too nested code by cleverly placing *break*, *continue* and *return* statements.

Continuing long statements: To continue a statement you can use backslashes in which case you should align the next line with the last dot or equal sign, or indent four spaces:

```
this_is_a_very_long(function_call, 'with many parameters') \
    .that_returns_an_object_with_an_attribute
```

```
MyModel.query.filter(MyModel.scalar > 120) \
    .order_by(MyModel.name.desc()) \
    .limit(10)
```

If you break in a statement with parentheses or braces, align to the braces:

```
this_is_a_very_long(function_call, 'with many parameters',
                        23, 42, 'and even more')
```

For lists or tuples with many items, break immediately after the opening brace:

```
items = [
    'this is the first', 'set of items', 'with more items',
    'to come in this line', 'like this'
]
```

Blank lines: Top level functions and classes are separated by two lines, everything else by one. Do not use too many blank lines to separate logical segments in code. Example:

```
def hello(name):
    print 'Hello %s!' % name

def goodbye(name):
    print 'See you %s.' % name

class MyClass(object):
    """This is a simple docstring"""

    def __init__(self, name):
        self.name = name

    def get_annoying_name(self):
        return self.name.upper() + '!!!!111'
```

23.2 Expressions and Statements

General whitespace rules:

- No whitespace for unary operators that are not words (e.g.: -, ~ etc.) as well on the inner side of parentheses.
- Whitespace is placed between binary operators.

Good:

```
exp = -1.05
value = (item_value / item_count) * offset / exp
value = my_list[index]
value = my_dict['key']
```

Bad:

```
exp = - 1.05
value = ( item_value / item_count ) * offset / exp
value = (item_value/item_count)*offset/exp
value=( item_value/item_count ) * offset/exp
value = my_list[ index ]
value = my_dict [ 'key' ]
```

Yoda statements are a no-go: Never compare constant with variable, always variable with constant:

Good:

```
if method == 'md5':  
    pass
```

Bad:

```
if 'md5' == method:  
    pass
```

Comparisons:

- against arbitrary types: `==` and `!=`
- against singletons with `is` and `is not` (eg: `foo is not None`)
- never compare something with `True` or `False` (for example never do `foo == False`, do not `foo` instead)

Negated containment checks: use `foo not in bar` instead of `not foo in bar`

Instance checks: `isinstance(a, C)` instead of `type(A) is C`, but try to avoid instance checks in general. Check for features.

23.3 Naming Conventions

- Class names: CamelCase, with acronyms kept uppercase (HTTPWriter and not HttpWriter)
- Variable names: lowercase_with_underscores
- Method and function names: lowercase_with_underscores
- Constants: UPPERCASE_WITH_UNDERSCORES
- precompiled regular expressions: `name_re`

Protected members are prefixed with a single underscore. Double underscores are reserved for mixin classes.

On classes with keywords, trailing underscores are appended. Clashes with builtins are allowed and **must not** be resolved by appending an underline to the variable name. If the function needs to access a shadowed builtin, rebind the builtin to a different name instead.

Function and method arguments:

- class methods: `cls` as first parameter
- instance methods: `self` as first parameter
- lambdas for properties might have the first parameter replaced with `x` like in `display_name = property(lambda x: x.real_name or x.username)`

23.4 Docstrings

Docstring conventions: All docstrings are formatted with reStructuredText as understood by Sphinx. Depending on the number of lines in the docstring, they are laid out differently. If it's just one line, the closing triple quote is on the same line as the opening, otherwise the text is on the same line as the opening quote and the triple quote that closes the string on its own line:

```
def foo():
    """This is a simple docstring"""

def bar():
    """This is a longer docstring with so much information in there
    that it spans three lines. In this case the closing triple quote
    is on its own line.
    """
```

Module header: The module header consists of an utf-8 encoding declaration (if non ASCII letters are used, but it is recommended all the time) and a standard docstring:

```
# -*- coding: utf-8 -*-
"""
    package.module
    ~~~~~

    A brief description goes here.

    :copyright: (c) YEAR by AUTHOR.
    :license: LICENSE_NAME, see LICENSE_FILE for more details.
"""
```

Please keep in mind that proper copyrights and license files are a requirement for approved Flask extensions.

23.5 Comments

Rules for comments are similar to docstrings. Both are formatted with reStructuredText. If a comment is used to document an attribute, put a colon after the opening pound sign (#):

```
class User(object):
    #: the name of the user as unicode string
    name = Column(String)
    #: the sha1 hash of the password + inline salt
    pw_hash = Column(String)
```

UPGRADING TO NEWER RELEASES

Flask itself is changing like any software is changing over time. Most of the changes are the nice kind, the kind where you don't have to change anything in your code to profit from a new release.

However every once in a while there are changes that do require some changes in your code or there are changes that make it possible for you to improve your own code quality by taking advantage of new features in Flask.

This section of the documentation enumerates all the changes in Flask from release to release and how you can change your code to have a painless updating experience.

If you want to use the *easy_install* command to upgrade your Flask installation, make sure to pass it the `-U` parameter:

```
$ easy_install -U Flask
```

24.1 Version 0.8

Flask introduced a new session interface system. We also noticed that there was a naming collision between *flask.session* the module that implements sessions and `flask.session` which is the global session object. With that introduction we moved the implementation details for the session system into a new module called `flask.sessions`. If you used the previously undocumented session support we urge you to upgrade.

If invalid JSON data was submitted Flask will now raise a `BadRequest` exception instead of letting the default `ValueError` bubble up. This has the advantage that you no longer have to handle that error to avoid an internal server error showing up for the user. If you were catching this down explicitly in the past as `ValueError` you will need to change this.

24.2 Version 0.7

In Flask 0.7 we cleaned up the code base internally a lot and did some backwards incompatible changes that make it easier to implement larger applications with Flask. Because we want to make upgrading as easy as possible we tried to counter the problems arising from these changes by providing a script that can ease the transition.

The script scans your whole application and generates an unified diff with changes it assumes are safe to apply. However as this is an automated tool it won't be able to find all use cases and it might miss some. We internally spread a lot of deprecation warnings all over the place to make it easy to find pieces of code that it was unable to upgrade.

We strongly recommend that you hand review the generated patchfile and only apply the chunks that look good.

If you are using git as version control system for your project we recommend applying the patch with `path -p1 < patchfile.diff` and then using the interactive commit feature to only apply the chunks that look good.

To apply the upgrade script do the following:

1. Download the script: [flask-07-upgrade.py](#)
2. Run it in the directory of your application:

```
python flask-07-upgrade.py > patchfile.diff
```

3. Review the generated patchfile.
4. Apply the patch:

```
patch -p1 < patchfile.diff
```

5. If you were using per-module template folders you need to move some templates around. Previously if you had a folder named `templates` next to a blueprint named `admin` the implicit template path automatically was `admin/index.html` for a template file called `templates/index.html`. This no longer is the case. Now you need to name the template `templates/admin/index.html`. The tool will not detect this so you will have to do that on your own.

Please note that deprecation warnings are disabled by default starting with Python 2.7. In order to see the deprecation warnings that might be emitted you have to enable them with the [warnings](#) module.

If you are working with windows and you lack the *patch* command line utility you can get it as part of various Unix runtime environments for windows including cygwin, msysgit or ming32. Also source control systems like svn, hg or git have builtin support for applying unified diffs as generated by the tool. Check the manual of your version control system for more information.

24.2.1 Bug in Request Locals

Due to a bug in earlier implementations the request local proxies now raise a `RuntimeError` instead of an `AttributeError` when they are unbound. If you caught these exceptions with `AttributeError` before, you should catch them with `RuntimeError` now.

Additionally the `send_file()` function is now issuing deprecation warnings if you depend on functionality that will be removed in Flask 1.0. Previously it was possible to use etags and mimetypes when file objects were passed. This was unreliable and caused issues for a few setups. If you get a deprecation warning, make sure to update your application to work with either filenames there or disable etag attaching and attach them yourself.

Old code:

```
return send_file(my_file_object)
return send_file(my_file_object)
```

New code:

```
return send_file(my_file_object, add_etags=False)
```

24.2.2 Upgrading to new Teardown Handling

We streamlined the behavior of the callbacks for request handling. For things that modify the response the `after_request()` decorators continue to work as expected, but for things that absolutely must happen at the end of request we introduced the new `teardown_request()` decorator. Unfortunately that change also made `after_request` work differently under error conditions. It's not consistently skipped if exceptions happen whereas previously it might have been called twice to ensure it is executed at the end of the request.

If you have database connection code that looks like this:

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

You are now encouraged to use this instead:

```
@app.teardown_request
def after_request(exception):
    g.db.close()
```

On the upside this change greatly improves the internal code flow and makes it easier to customize the dispatching and error handling. This makes it now a lot easier to write unit tests as you can prevent closing down of database connections for a while. You can take advantage of the fact that the `teardown` callbacks are called when the re-

sponse context is removed from the stack so a test can query the database after request handling:

```
with app.test_client() as client:
    resp = client.get('/')
    # g.db is still bound if there is such a thing

# and here it's gone
```

24.2.3 Manual Error Handler Attaching

While it is still possible to attach error handlers to `Flask.error_handlers` it's discouraged to do so and in fact deprecated. In general we no longer recommend custom error handler attaching via assignments to the underlying dictionary due to the more complex internal handling to support arbitrary exception classes and blueprints. See `Flask.errorhandler()` for more information.

The proper upgrade is to change this:

```
app.error_handlers[403] = handle_error
```

Into this:

```
app.register_error_handler(403, handle_error)
```

Alternatively you should just attach the function with a decorator:

```
@app.errorhandler(403)
def handle_error(e):
    ...
```

(Note that `register_error_handler()` is new in Flask 0.7)

24.2.4 Blueprint Support

Blueprints replace the previous concept of “Modules” in Flask. They provide better semantics for various features and work better with large applications. The update script provided should be able to upgrade your applications automatically, but there might be some cases where it fails to upgrade. What changed?

- Blueprints need explicit names. Modules had an automatic name guessing scheme where the shortname for the module was taken from the last part of the import module. The upgrade script tries to guess that name but it might fail as this information could change at runtime.
- Blueprints have an inverse behavior for `url_for()`. Previously `.foo` told `url_for()` that it should look for the endpoint `foo` on the application. Now it means “relative to current module”. The script will inverse all calls to `url_for()` automatically for you. It will do this in a very eager way so you might end up with some unnecessary leading dots in your code if you're not using modules.

- Blueprints do not automatically provide static folders. They will also no longer automatically export templates from a folder called *templates* next to their location however but it can be enabled from the constructor. Same with static files: if you want to continue serving static files you need to tell the constructor explicitly the path to the static folder (which can be relative to the blueprint's module path).
- Rendering templates was simplified. Now the blueprints can provide template folders which are added to a general template searchpath. This means that you need to add another subfolder with the blueprint's name into that folder if you want `blueprintname/template.html` as the template name.

If you continue to use the *Module* object which is deprecated, Flask will restore the previous behavior as good as possible. However we strongly recommend upgrading to the new blueprints as they provide a lot of useful improvement such as the ability to attach a blueprint multiple times, blueprint specific error handlers and a lot more.

24.3 Version 0.6

Flask 0.6 comes with a backwards incompatible change which affects the order of after-request handlers. Previously they were called in the order of the registration, now they are called in reverse order. This change was made so that Flask behaves more like people expected it to work and how other systems handle request pre- and postprocessing. If you depend on the order of execution of post-request functions, be sure to change the order.

Another change that breaks backwards compatibility is that context processors will no longer override values passed directly to the template rendering function. If for example *request* is as variable passed directly to the template, the default context processor will not override it with the current request object. This makes it easier to extend context processors later to inject additional variables without breaking existing template not expecting them.

24.4 Version 0.5

Flask 0.5 is the first release that comes as a Python package instead of a single module. There were a couple of internal refactoring so if you depend on undocumented internal details you probably have to adapt the imports.

The following changes may be relevant to your application:

- autoescaping no longer happens for all templates. Instead it is configured to only happen on files ending with `.html`, `.htm`, `.xml` and `.xhtml`. If you have templates with different extensions you should override the `select_jinja_autoescape()` method.
- Flask no longer supports zipped applications in this release. This functionality might come back in future releases if there is demand for this feature. Removing

support for this makes the Flask internal code easier to understand and fixes a couple of small issues that make debugging harder than necessary.

- The `create_jinja_loader` function is gone. If you want to customize the Jinja loader now, use the `create_jinja_environment()` method instead.

24.5 Version 0.4

For application developers there are no changes that require changes in your code. In case you are developing on a Flask extension however, and that extension has a unittest-mode you might want to link the activation of that mode to the new TESTING flag.

24.6 Version 0.3

Flask 0.3 introduces configuration support and logging as well as categories for flashing messages. All these are features that are 100% backwards compatible but you might want to take advantage of them.

24.6.1 Configuration Support

The configuration support makes it easier to write any kind of application that requires some sort of configuration. (Which most likely is the case for any application out there).

If you previously had code like this:

```
app.debug = DEBUG
app.secret_key = SECRET_KEY
```

You no longer have to do that, instead you can just load a configuration into the config object. How this works is outlined in *Configuration Handling*.

24.6.2 Logging Integration

Flask now configures a logger for you with some basic and useful defaults. If you run your application in production and want to profit from automatic error logging, you might be interested in attaching a proper log handler. Also you can start logging warnings and errors into the logger when appropriately. For more information on that, read *Handling Application Errors*.

24.6.3 Categories for Flash Messages

Flash messages can now have categories attached. This makes it possible to render errors, warnings or regular messages differently for example. This is an opt-in feature because it requires some rethinking in the code.

Read all about that in the *Message Flashing* pattern.

FLASK CHANGELOG

Here you can see the full list of changes between each Flask release.

25.1 Version 0.8

Release date to be decided, codename to be chosen.

- Refactored session support into a session interface so that the implementation of the sessions can be changed without having to override the Flask class.
- Empty session cookies are now deleted properly automatically.
- View functions can now opt out of getting the automatic OPTIONS implementation.
- HTTP exceptions and Bad Request errors can now be trapped so that they show up normally in the traceback.
- Flask in debug mode is now detecting some common problems and tries to warn you about them.
- Flask in debug mode will now complain with an assertion error if a view was attached after the first request was handled. This gives earlier feedback when users forget to import view code ahead of time.
- Added the ability to register callbacks that are only triggered once at the beginning of the first request. (`Flask.before_first_request()`)
- Malformed JSON data will now trigger a bad request HTTP exception instead of a value error which usually would result in a 500 internal server error if not handled. This is a backwards incompatible change.

25.2 Version 0.7.3

Bugfix release, release date to be decided

- Fixed the Jinja2 environment's `list_templates` method not returning the correct names when blueprints or modules were involved.

25.3 Version 0.7.2

Bugfix release, released on July 6th 2011

- Fixed an issue with URL processors not properly working on blueprints.

25.4 Version 0.7.1

Bugfix release, released on June 29th 2011

- Added missing future import that broke 2.5 compatibility.
- Fixed an infinite redirect issue with blueprints.

25.5 Version 0.7

Released on June 28th 2011, codename Grappa

- Added `make_default_options_response()` which can be used by subclasses to alter the default behaviour for *OPTIONS* responses.
- Unbound locals now raise a proper `RuntimeError` instead of an `AttributeError`.
- Mimetype guessing and etag support based on file objects is now deprecated for `flask.send_file()` because it was unreliable. Pass filenames instead or attach your own etags and provide a proper mimetype by hand.
- Static file handling for modules now requires the name of the static folder to be supplied explicitly. The previous autodetection was not reliable and caused issues on Google's App Engine. Until 1.0 the old behaviour will continue to work but issue dependency warnings.
- fixed a problem for Flask to run on jython.
- added a *PROPAGATE_EXCEPTIONS* configuration variable that can be used to flip the setting of exception propagation which previously was linked to *DEBUG* alone and is now linked to either *DEBUG* or *TESTING*.
- Flask no longer internally depends on rules being added through the *add_url_rule* function and can now also accept regular werkzeug rules added to the url map.
- Added an *endpoint* method to the flask application object which allows one to register a callback to an arbitrary endpoint with a decorator.
- Use Last-Modified for static file sending instead of Date which was incorrectly introduced in 0.6.
- Added *create_jinja_loader* to override the loader creation process.
- Implemented a silent flag for *config.from_pyfile*.

- Added *teardown_request* decorator, for functions that should run at the end of a request regardless of whether an exception occurred. Also the behavior for *after_request* was changed. It's now no longer executed when an exception is raised. See *Upgrading to new Teardown Handling*
- Implemented `flask.has_request_context()`
- Deprecated *init_jinja_globals*. Override the `create_jinja_environment()` method instead to achieve the same functionality.
- Added `flask.safe_join()`
- The automatic JSON request data unpacking now looks at the `charset_mimetype` parameter.
- Don't modify the session on `flask.get_flashed_messages()` if there are no messages in the session.
- *before_request* handlers are now able to abort requests with errors.
- it is not possible to define user exception handlers. That way you can provide custom error messages from a central hub for certain errors that might occur during request processing (for instance database connection errors, timeouts from remote resources etc.).
- Blueprints can provide blueprint specific error handlers.
- Implemented generic *Pluggable Views* (class based views).

25.6 Version 0.6.1

Bugfix release, released on December 31st 2010

- Fixed an issue where the default *OPTIONS* response was not exposing all valid methods in the *Allow* header.
- Jinja2 template loading syntax now allows `"/` in front of a template load path. Previously this caused issues with module setups.
- Fixed an issue where the subdomain setting for modules was ignored for the static folder.
- Fixed a security problem that allowed clients to download arbitrary files if the host server was a windows based operating system and the client uses backslashes to escape the directory the files where exposed from.

25.7 Version 0.6

Released on July 27th 2010, codename Whisky

- after request functions are now called in reverse order of registration.

- OPTIONS is now automatically implemented by Flask unless the application explicitly adds 'OPTIONS' as method to the URL rule. In this case no automatic OPTIONS handling kicks in.
- static rules are now even in place if there is no static folder for the module. This was implemented to aid GAE which will remove the static folder if it's part of a mapping in the .yaml file.
- the config is now available in the templates as *config*.
- context processors will no longer override values passed directly to the render function.
- added the ability to limit the incoming request data with the new MAX_CONTENT_LENGTH configuration value.
- the endpoint for the flask.Module.add_url_rule() method is now optional to be consistent with the function of the same name on the application object.
- added a flask.make_response() function that simplifies creating response object instances in views.
- added signalling support based on blinker. This feature is currently optional and supposed to be used by extensions and applications. If you want to use it, make sure to have [blinker](#) installed.
- refactored the way URL adapters are created. This process is now fully customizable with the create_url_adapter() method.
- modules can now register for a subdomain instead of just an URL prefix. This makes it possible to bind a whole module to a configurable subdomain.

25.8 Version 0.5.2

Bugfix Release, released on July 15th 2010

- fixed another issue with loading templates from directories when modules were used.

25.9 Version 0.5.1

Bugfix Release, released on July 6th 2010

- fixes an issue with template loading from directories when modules were used.

25.10 Version 0.5

Released on July 6th 2010, codename Calvados

- fixed a bug with subdomains that was caused by the inability to specify the server name. The server name can now be set with the *SERVER_NAME* config key. This key is now also used to set the session cookie cross-subdomain wide.
- autoescaping is no longer active for all templates. Instead it is only active for *.html*, *.htm*, *.xml* and *.xhtml*. Inside templates this behaviour can be changed with the *autoescape* tag.
- refactored Flask internally. It now consists of more than a single file.
- *flask.send_file()* now emits etags and has the ability to do conditional responses builtin.
- (temporarily) dropped support for zipped applications. This was a rarely used feature and led to some confusing behaviour.
- added support for per-package template and static-file directories.
- removed support for *create_jinja_loader* which is no longer used in 0.5 due to the improved module support.
- added a helper function to expose files from any directory.

25.11 Version 0.4

Released on June 18th 2010, codename Rakia

- added the ability to register application wide error handlers from modules.
- *after_request()* handlers are now also invoked if the request dies with an exception and an error handling page kicks in.
- test client has not the ability to preserve the request context for a little longer. This can also be used to trigger custom requests that do not pop the request stack for testing.
- because the Python standard library caches loggers, the name of the logger is configurable now to better support unittests.
- added *TESTING* switch that can activate unittesting helpers.
- the logger switches to *DEBUG* mode now if debug is enabled.

25.12 Version 0.3.1

Bugfix release, released on May 28th 2010

- fixed a error reporting bug with *flask.Config.from_envvar()*
- removed some unused code from flask
- release does no longer include development leftover files (*.git* folder for themes, built documentation in zip and pdf file and some *.pyc* files)

25.13 Version 0.3

Released on May 28th 2010, codename Schnaps

- added support for categories for flashed messages.
- the application now configures a `logging.Handler` and will log request handling exceptions to that logger when not in debug mode. This makes it possible to receive mails on server errors for example.
- added support for context binding that does not require the use of the with statement for playing in the console.
- the request context is now available within the with statement making it possible to further push the request context or pop it.
- added support for configurations.

25.14 Version 0.2

Released on May 12th 2010, codename Jägermeister

- various bugfixes
- integrated JSON support
- added `get_template_attribute()` helper function.
- `add_url_rule()` can now also register a view function.
- refactored internal request dispatching.
- server listens on 127.0.0.1 by default now to fix issues with chrome.
- added external URL support.
- added support for `send_file()`
- module support and internal request handling refactoring to better support pluggable applications.
- sessions can be set to be permanent now on a per-session basis.
- better error reporting on missing secret keys.
- added support for Google Appengine.

25.15 Version 0.1

First public preview release.

LICENSE

Flask is licensed under a three clause BSD License. It basically means: do whatever you want with it as long as the copyright in Flask sticks around, the conditions are not modified and the disclaimer is present. Furthermore you must not use the names of the authors to promote derivatives of the software without written consent.

The full license text can be found below (*Flask License*). For the documentation and artwork different licenses apply.

26.1 Authors

Flask is written and maintained by Armin Ronacher and various contributors:

26.1.1 Development Lead

- Armin Ronacher <armin.ronacher@active-4.com>

26.1.2 Patches and Suggestions

- Adam Zapletal
- Ali Afshar
- Chris Edgemon
- Chris Grindstaff
- Christopher Grebs
- Florent Xicluna
- Georg Brandl
- Justin Quick
- Kenneth Reitz
- Marian Sigler

- Matt Campell
- Matthew Frazier
- Michael van Tellingen
- Ron DuPlain
- Sebastien Estienne
- Simon Sapin
- Stephane Wirtel
- Thomas Schranz
- Zhao Xiaohong

26.2 General License Definitions

The following section contains the full license texts for Flask and the documentation.

- “AUTHORS” hereby refers to all the authors listed in the *Authors* section.
- The “*Flask License*” applies to all the sourcecode shipped as part of Flask (Flask itself as well as the examples and the unittests) as well as documentation.
- The “*Flask Artwork License*” applies to the project’s Horn-Logo.

26.3 Flask License

Copyright (c) 2010 by Armin Ronacher and contributors. See AUTHORS for more details.

Some rights reserved.

Redistribution and use in source and binary forms of the software as well as documentation, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE AND DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

26.4 Flask Artwork License

Copyright (c) 2010 by Armin Ronacher.

Some rights reserved.

This logo or a modified version may be used by anyone to refer to the Flask project, but does not indicate endorsement by the project.

Redistribution and use in source (the SVG file) and binary forms (rendered PNG files etc.) of the image, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice and this list of conditions.
- The names of the contributors to the Flask software (see AUTHORS) may not be used to endorse or promote products derived from this software without specific prior written permission.

Note: we would appreciate that you make the image a link to <http://flask.pocoo.org/> if you use it on a web page.

INDEX

Symbols

`_request_ctx_stack` (in module flask), 179

A

`abort()` (in module flask), 170

`add_url_rule()` (flask.Blueprint method), 160

`add_url_rule()`
(flask.blueprints.BlueprintSetupState method), 180

`add_url_rule()` (flask.Flask method), 146

`after_app_request()` (flask.Blueprint method), 160

`after_request()` (flask.Blueprint method), 160

`after_request()` (flask.Flask method), 147

`after_request_funcs` (flask.Flask attribute), 147

`app` (flask.blueprints.BlueprintSetupState attribute), 180

`app_context_processor()` (flask.Blueprint method), 160

`app_errorhandler()` (flask.Blueprint method), 160

`app_url_defaults()` (flask.Blueprint method), 161

`app_url_value_preprocessor()`
(flask.Blueprint method), 161

`args` (flask.Request attribute), 163

`as_view()` (flask.views.View class method), 182

B

`base_url` (flask.Request attribute), 164

`before_app_first_request()`
(flask.Blueprint method), 161

`before_app_request()` (flask.Blueprint method), 161

`before_first_request()` (flask.Flask method), 147

`before_first_request_funcs` (flask.Flask attribute), 147

`before_request()` (flask.Blueprint method), 161

`before_request()` (flask.Flask method), 147

`before_request_funcs` (flask.Flask attribute), 147

`Blueprint` (class in flask), 160

`blueprint` (flask.blueprints.BlueprintSetupState attribute), 180

`blueprint` (flask.Request attribute), 164

`blueprints` (flask.Flask attribute), 147

`BlueprintSetupState` (class in flask.blueprints), 180

C

`Config` (class in flask), 177

`config` (flask.Flask attribute), 147

`context_processor()` (flask.Blueprint method), 161

`context_processor()` (flask.Flask method), 148

`cookies` (flask.Request attribute), 163

`create_global_jinja_loader()` (flask.Flask method), 148

`create_jinja_environment()` (flask.Flask method), 148

`create_url_adapter()` (flask.Flask method), 148

`current_app` (in module flask), 169

D

`data` (flask.Request attribute), 163

`data` (flask.Response attribute), 166

`debug` (flask.Flask attribute), 148

`debug_log_format` (flask.Flask attribute), 148
`default_config` (flask.Flask attribute), 148
`dispatch_request()` (flask.Flask method), 148
`dispatch_request()` (flask.views.View method), 182
`do_teardown_request()` (flask.Flask method), 148
E
`enable_modules` (flask.Flask attribute), 149
`endpoint` (flask.Request attribute), 164
`endpoint()` (flask.Blueprint method), 161
`endpoint()` (flask.Flask method), 149
`environ` (flask.Request attribute), 164
environment variable
 `FLASKR_SETTINGS`, 30
 `YOURAPPLICATION_SETTINGS`, 57
`error_handler_spec` (flask.Flask attribute), 149
`errorhandler()` (flask.Blueprint method), 161
`errorhandler()` (flask.Flask method), 149
`escape()` (flask.Markup class method), 174
`escape()` (in module flask), 173
`extensions` (flask.Flask attribute), 150
F
`files` (flask.Request attribute), 163
`first_registration`
 (flask.blueprints.BlueprintSetupState attribute), 180
`flash()` (in module flask), 175
`Flask` (class in flask), 145
`flask` (module), 145
`flask.signals.Namespace` (built-in class), 181
`FLASKR_SETTINGS`, 30
`form` (flask.Request attribute), 163
`from_envvar()` (flask.Config method), 178
`from_object()` (flask.Config method), 178
`from_pyfile()` (flask.Config method), 178
`full_dispatch_request()` (flask.Flask method), 150
G
`g` (in module flask), 169
`get_cookie_domain()`
 (flask.sessions.SessionInterface method), 168
`get_expiration_time()`
 (flask.sessions.SessionInterface method), 168
`get_flashed_messages()` (in module flask), 175
`get_template_attribute()` (in module flask), 177
`got_first_request` (flask.Flask attribute), 150
`got_request_exception` (in module flask), 181
H
`handle_exception()` (flask.Flask method), 150
`handle_http_exception()` (flask.Flask method), 150
`handle_user_exception()` (flask.Flask method), 150
`has_request_context()` (in module flask), 170
`has_static_folder` (flask.Blueprint attribute), 161
`has_static_folder` (flask.Flask attribute), 150
`headers` (flask.Request attribute), 163
`headers` (flask.Response attribute), 166
I
`init_jinja_globals()` (flask.Flask method), 150
`inject_url_defaults()` (flask.Flask method), 151
`is_null_session()`
 (flask.sessions.SessionInterface method), 168
`is_xhr` (flask.Request attribute), 164
J
`jinja_env` (flask.Flask attribute), 151
`jinja_loader` (flask.Blueprint attribute), 161
`jinja_loader` (flask.Flask attribute), 151
`jinja_options` (flask.Flask attribute), 151
`json` (flask.Request attribute), 164, 165

json (in module flask), 176
jsonify() (in module flask), 175

L

logger (flask.Flask attribute), 151
logger_name (flask.Flask attribute), 151

M

make_default_options_response()
 (flask.Flask method), 151
make_null_session() (flask.Flask method),
 151
make_null_session()
 (flask.sessions.SessionInterface
 method), 168
make_response() (flask.Flask method),
 151
make_response() (in module flask), 171
make_setup_state() (flask.Blueprint
 method), 161
Markup (class in flask), 173
match_request()
 (flask.ctx.RequestContext
 method), 179
max_content_length (flask.Request at-
 tribute), 165
method (flask.Request attribute), 164
MethodView (class in flask.views), 182
mimetype (flask.Response attribute), 166
modified (flask.session attribute), 167
modified (flask.sessions.SessionMixin at-
 tribute), 169
module (flask.Request attribute), 165

N

new (flask.session attribute), 167
new (flask.sessions.SessionMixin at-
 tribute), 169
null_session_class
 (flask.sessions.SessionInterface
 attribute), 168
NullSession (class in flask.sessions), 169

O

on_json_loading_failed() (flask.Request
 method), 165
open_resource() (flask.Blueprint method),
 161

open_resource() (flask.Flask method), 152
open_session() (flask.Flask method), 152
open_session()
 (flask.sessions.SessionInterface
 method), 168
options (flask.blueprints.BlueprintSetupState
 attribute), 180

P

path (flask.Request attribute), 164
permanent (flask.session attribute), 167
permanent (flask.sessions.SessionMixin
 attribute), 169
permanent_session_lifetime (flask.Flask
 attribute), 152
pop() (flask.ctx.RequestContext method),
 179
preprocess_request() (flask.Flask
 method), 152
preserve_context_on_exception
 (flask.Flask attribute), 153
process_response() (flask.Flask method),
 153
propagate_exceptions (flask.Flask at-
 tribute), 153
push() (flask.ctx.RequestContext
 method), 179
Python Enhancement Proposals
 PEP 8, 211

R

record() (flask.Blueprint method), 162
record_once() (flask.Blueprint method),
 162
redirect() (in module flask), 170
register() (flask.Blueprint method), 162
register_blueprint() (flask.Flask method),
 153
register_error_handler() (flask.Flask
 method), 153
register_module() (flask.Flask method),
 153
render_template() (in module flask), 176
render_template_string() (in module
 flask), 176
Request (class in flask), 163
request (class in flask), 165
request_class (flask.Flask attribute), 153

request_context() (flask.Flask method), 153
 request_finished (in module flask), 181
 request_started (in module flask), 181
 request_tearing_down (in module flask), 181
 RequestContext (class in flask.ctx), 179
 Response (class in flask), 165
 response_class (flask.Flask attribute), 154
 route() (flask.Blueprint method), 162
 route() (flask.Flask method), 154
 routing_exception (flask.Request attribute), 165
 run() (flask.Flask method), 155

S

safe_join() (in module flask), 173
 save_session() (flask.Flask method), 156
 save_session() (flask.sessions.SessionInterface method), 168
 script_root (flask.Request attribute), 164
 secret_key (flask.Flask attribute), 156
 SecureCookieSessionInterface (class in flask.sessions), 168
 select_jinja_autoescape() (flask.Flask method), 156
 send_file() (in module flask), 171
 send_from_directory() (in module flask), 172
 send_static_file() (flask.Blueprint method), 162
 send_static_file() (flask.Flask method), 156
 session (class in flask), 167
 session_class (flask.sessions.SecureCookieSessionInterface attribute), 169
 session_cookie_name (flask.Flask attribute), 156
 session_interface (flask.Flask attribute), 156
 SessionInterface (class in flask.sessions), 167
 SessionMixin (class in flask.sessions), 169
 set_cookie() (flask.Response method), 166
 signal() (flask.signals.Namespace method), 181
 signals_available (in module flask), 181
 status_code (flask.Response attribute), 166
 stream (flask.Request attribute), 163
 striptags() (flask.Markup method), 174
 subdomain (flask.blueprints.BlueprintSetupState attribute), 180

T

teardown_app_request() (flask.Blueprint method), 162
 teardown_request() (flask.Blueprint method), 162
 teardown_request() (flask.Flask method), 156
 teardown_request_funcs (flask.Flask attribute), 157
 template_context_processors (flask.Flask attribute), 157
 template_filter() (flask.Flask method), 157
 template_rendered (in module flask), 181
 test_client() (flask.Flask method), 157
 test_client_class (flask.Flask attribute), 158
 test_request_context() (flask.Flask method), 158
 testing (flask.Flask attribute), 158
 trap_http_exception() (flask.Flask method), 158

U

unescape() (flask.Markup method), 174
 update_template_context() (flask.Flask method), 158
 url (flask.Request attribute), 164
 url_default_functions (flask.Flask attribute), 158
 url_defaults (flask.blueprints.BlueprintSetupState attribute), 180
 url_defaults() (flask.Blueprint method), 163
 url_defaults() (flask.Flask method), 159
 url_for() (in module flask), 170
 url_map (flask.Flask attribute), 159
 url_prefix (flask.blueprints.BlueprintSetupState attribute), 181
 url_root (flask.Request attribute), 164
 url_rule (flask.Request attribute), 165

`url_rule_class` (flask.Flask attribute), 159
`url_value_preprocessor()` (flask.Blueprint method), 163
`url_value_preprocessor()` (flask.Flask method), 159
`url_value_preprocessors` (flask.Flask attribute), 159
`use_x_sendfile` (flask.Flask attribute), 159

V

`values` (flask.Request attribute), 163
`View` (class in flask.views), 182
`view_args` (flask.Request attribute), 165
`view_functions` (flask.Flask attribute), 159

W

`wsgi_app()` (flask.Flask method), 159

Y

`YOURAPPLICATION_SETTINGS`, 57