Flask

web development,
one drop at a time

Flask Documentation

Release 0.13.dev

Apr 12, 2018

Contents

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.

1.1 Foreword

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

1.1.1 What does "micro" mean?

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

By default, Flask does not include a database abstraction layer, form validation or anything else where different libraries already exist that can handle that. Instead, Flask supports extensions to add such functionality to your application as if it was implemented in Flask itself. Numerous extensions provide database integration, form val-

idation, upload handling, various open authentication technologies, and more. Flask may be "micro", but it's ready for production use on a variety of needs.

1.1.2 Configuration and Conventions

Flask has many configuration values, with sensible defaults, and a few conventions when getting started. By convention, templates and static files are stored in subdirectories within the application's Python source tree, with the names `templates` and `static` respectively. While this can be changed, you usually don't have to, especially when getting started.

1.1.3 Growing with Flask

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

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

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

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

1.2 Foreword for Experienced Programmers

1.2.1 Thread-Locals in Flask

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

1.2.2 Develop for the Web with Caution

Always keep security in mind when building web applications.

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

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

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

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

1.3 Installation

1.3.1 Python Version

We recommend using the latest version of Python 3. Flask supports Python 3.4 and newer, Python 2.7, and PyPy.

1.3.2 Dependencies

These distributions will be installed automatically when installing Flask.

- Werkzeug implements WSGI, the standard Python interface between applications and servers.

- Jinja is a template language that renders the pages your application serves.

- MarkupSafe comes with Jinja. It escapes untrusted input when rendering templates to avoid injection attacks.

- ItsDangerous securely signs data to ensure its integrity. This is used to protect Flask's session cookie.

- Click is a framework for writing command line applications. It provides the `flask` command and allows adding custom management commands.

Optional dependencies

These distributions will not be installed automatically. Flask will detect and use them if you install them.

- Blinker provides support for *Signals*.

- SimpleJSON is a fast JSON implementation that is compatible with Python's `json` module. It is preferred for JSON operations if it is installed.

- python-dotenv enables support for *Environment Variables From dotenv* when running `flask` commands.

- Watchdog provides a faster, more efficient reloader for the development server.

1.3.3 Virtual environments

Use a virtual environment to manage the dependencies for your project, both in development and in production.

What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries, or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

Virtual environments are independent groups of Python libraries, one for each project. Packages installed for one project will not affect other projects or the operating system's packages.

Python 3 comes bundled with the `venv` module to create virtual environments. If you're using a modern version of Python, you can continue on to the next section.

If you're using Python 2, see *Install virtualenv* first.

Create an environment

Create a project folder and a `venv` folder within:

```
mkdir myproject
cd myproject
python3 -m venv venv
```

On Windows:

```
py -3 -m venv venv
```

If you needed to install virtualenv because you are on an older version of Python, use the following command instead:

```
virtualenv venv
```

On Windows:

```
\Python27\Scripts\virtualenv.exe venv
```

Activate the environment

Before you work on your project, activate the corresponding environment:

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

On Windows:

```
venv\Scripts\activate
```

Your shell prompt will change to show the name of the activated environment.

1.3.4 Install Flask

Within the activated environment, use the following command to install Flask:

```
pip install Flask
```

Living on the edge

If you want to work with the latest Flask code before it's released, install or update the code from the master branch:

```
pip install -U https://github.com/pallets/flask/archive/master.tar.gz
```

1.3.5 Install virtualenv

If you are using Python 2, the venv module is not available. Instead, install virtualenv.

On Linux, virtualenv is provided by your package manager:

```
# Debian, Ubuntu
sudo apt-get install python-virtualenv

# CentOS, Fedora
sudo yum install python-virtualenv

# Arch
sudo pacman -S python-virtualenv
```

If you are on Mac OS X or Windows, download get-pip.py, then:

```
sudo python2 Downloads/get-pip.py
sudo python2 -m pip install virtualenv
```

On Windows, as an administrator:

```
\Python27\python.exe Downloads\get-pip.py
\Python27\python.exe -m pip install virtualenv
```

Now you can continue to *Create an environment*.

1.4 Quickstart

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

1.4.1 A Minimal Application

A minimal Flask application looks something like this:

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


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

So what did that code do?

1. First we imported the `Flask` class. An instance of this class will be our WSGI application.

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

3. We then use the `route()` decorator to tell Flask what URL should trigger our function.

4. The function is given a name which is also used to generate URLs for that particular function, and returns the message we want to display in the user's browser.

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

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

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

If you are on Windows, the environment variable syntax depends on command line interpreter. On Command Prompt:

```
C:\path\to\app>set FLASK_APP=hello.py
```

And on PowerShell:

```
PS C:\path\to\app> $env:FLASK_APP = "hello.py"
```

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

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

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

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

Externally Visible Server

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

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

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

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

1.4.2 What to do if the Server does not Start

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

Old Version of Flask

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

Invalid Import Name

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

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

1.4.3 Debug Mode

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

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

To enable all development features (including debug mode) you can export the FLASK_ENV environment variable and set it to development before running the server:

```
$ export FLASK_ENV=development
$ flask run
```

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

This does the following things:

1. it activates the debugger

2. it activates the automatic reloader

3. it enables the debug mode on the Flask application.

You can also control debug mode separately from the environment by exporting FLASK_DEBUG=1.

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

Attention

Even though the interactive debugger does not work in forking environments (which makes it nearly impossible to use on production servers), it still allows the execution

---

of arbitrary code. This makes it a major security risk and therefore it **must never be used on production machines**.

Screenshot of the debugger in action:
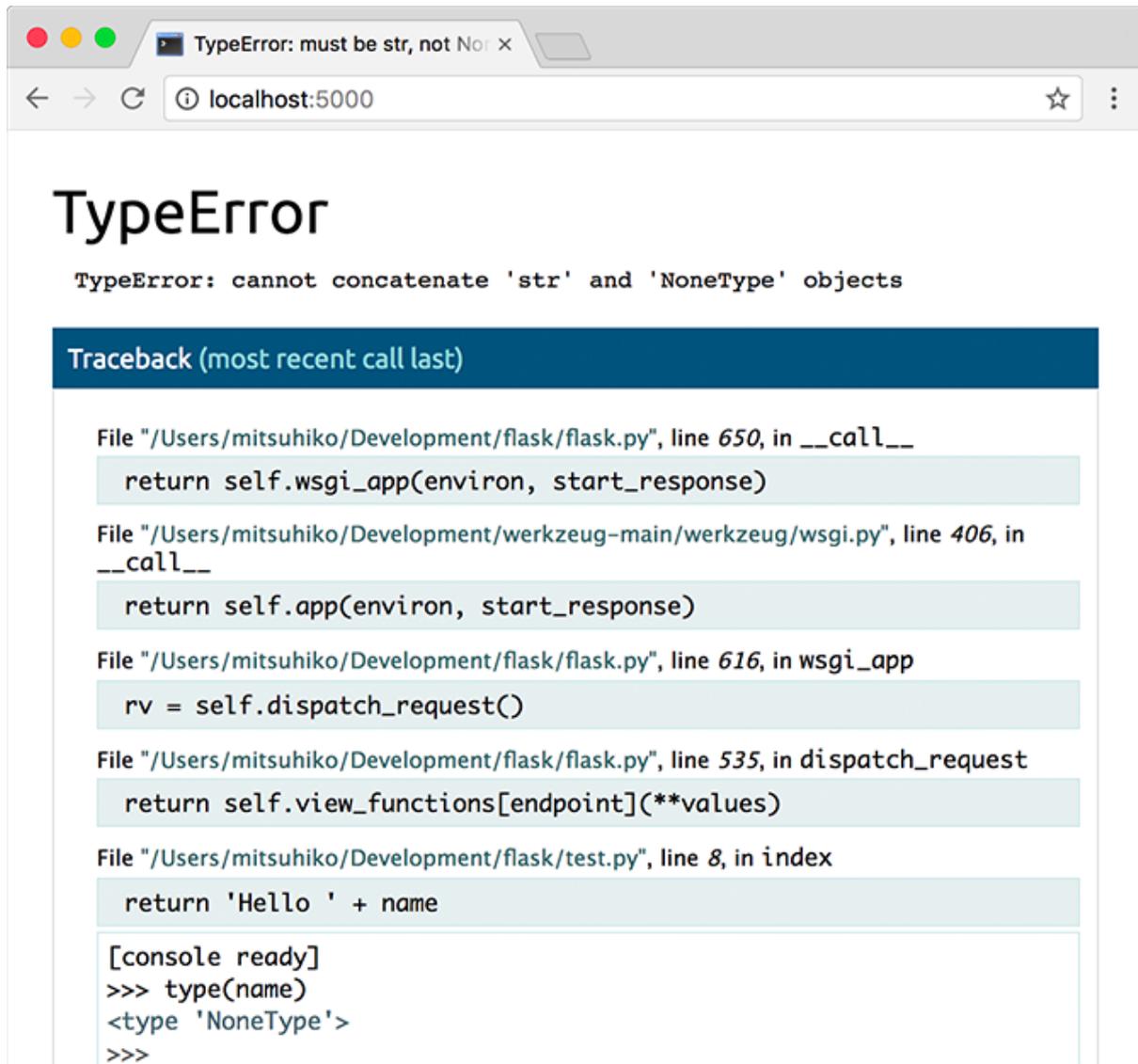
More information on using the debugger can be found in the Werkzeug documentation.

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

1.4.4 Routing

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to directly visit a page.

Use the `route()` decorator to bind a function to a URL.

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

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

You can do more! You can make parts of the URL dynamic and attach multiple rules to a function.

Variable Rules

You can add variable sections to a URL by marking sections with `<variable_name>`. Your function then receives the `<variable_name>` as a keyword argument. Optionally, you can use a converter to specify the type of the argument like `<converter:variable_name>`.

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

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

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return 'Subpath %s' % subpath
```

Converter types:

string	(default) accepts any text without a slash
int	accepts positive integers
float	accepts positive floating point values
path	like string but also accepts slashes
uuid	accepts UUID strings

Unique URLs / Redirection Behavior

The following two rules differ in their use of a trailing slash.

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

```
    return 'The project page'

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

The canonical URL for the `projects` endpoint has a trailing slash. It's similar to a folder in a file system. If you access the URL without a trailing slash, Flask redirects you to the canonical URL with the trailing slash.

The canonical URL for the about endpoint does not have a trailing slash. It's similar to the pathname of a file. Accessing the URL with a trailing slash produces a 404 "Not Found" error. This helps keep URLs unique for these resources, which helps search engines avoid indexing the same page twice.

URL Building

To build a URL to a specific function, use the `url_for()` function. It accepts the name of the function as its first argument and any number of keyword arguments, each corresponding to a variable part of the URL rule. Unknown variable parts are appended to the URL as query parameters.

Why would you want to build URLs using the URL reversing function `url_for()` instead of hard-coding them into your templates?

1. Reversing is often more descriptive than hard-coding the URLs.

2. **You can change your URLs in one go instead of needing to remember to** manually change hard-coded URLs.

3. **URL building handles escaping of special characters and Unicode data** transparently.

4. The generated paths are always absolute, avoiding unexpected behavior of relative paths in browsers.

5. **If your application is placed outside the URL root, for example, in** `/myapplication` instead of `/`, `url_for()` properly handles that for you.

For example, here we use the `test_request_context()` method to try out `url_for()`. `test_request_context()` tells Flask to behave as though it's handling a request even while we use a Python shell. See *Context Locals*.

```
from flask import Flask, url_for

app = Flask(__name__)

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

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

```
def login():
    return 'login'

@app.route('/user/<username>')
def profile(username):
    return '{}'s profile'.format(username)

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
```

HTTP Methods

Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to GET requests. You can use the methods argument of the route() decorator to handle different HTTP methods.

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

If GET is present, Flask automatically adds support for the HEAD method and handles HEAD requests according to the the [HTTP RFC](). Likewise, OPTIONS is automatically implemented for you.

1.4.5 Static Files

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

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

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

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

1.4.6 Rendering Templates

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

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

```python
from flask import render_template

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

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

Case 1: a module:

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

Case 2: a package:

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

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

Here is an example template:

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

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

---

[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.

the `get_flashed_messages()` function.

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

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

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

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

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

1.4.7 Accessing Request Data

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

Context Locals

Insider Information

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

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

Imagine the context being the handling thread. A request comes in and the web server decides to spawn a new thread (or something else, the underlying object is capable

of dealing with concurrency systems other than threads). When Flask starts its internal request handling it figures out that the current thread is the active context and binds the current application and the WSGI environments to that context (thread). It does that in an intelligent way so that one application can invoke another application without breaking.

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

```python
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:

```python
from flask import request

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

The Request Object

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

```python
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:

```python
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
```

```
    else:
        error = 'Invalid username/password'
# the code below is executed if the request method
# was GET or the credentials were invalid
return render_template('login.html', error=error)
```

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

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

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

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

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

File Uploads

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

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

```
from flask import request

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

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

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

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

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

Cookies

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

Reading cookies:

```python
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:

```python
from flask import make_response

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

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

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

For this also see *About Responses*.

1.4.8 Redirects and Errors

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

```python
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:

```python
from flask import render_template

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

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

See *Error handlers* for more details.

1.4.9 About Responses

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

1. If a response object of the correct type is returned it's directly returned from the view.

2. If it's a string, a response object is created with that data and the default parameters.

3. If a tuple is returned the items in the tuple can provide extra information. Such tuples have to be in the form (`response, status, headers`) or (`response, headers`) where at least one item has to be in the tuple. The `status` value will

override the status code and `headers` can be a list or dictionary of additional header values.

4. If none of that works, Flask will assume the return value is a valid WSGI application and convert that into a response object.

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

Imagine you have a view like this:

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

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

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

1.4.10 Sessions

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

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

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

app = Flask(__name__)

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

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

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
```

```
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''


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

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

How to generate good secret keys

A secret key should be as random as possible. Your operating system has ways to generate pretty random data based on a cryptographic random generator. Use the following command to quickly generate a value for Flask.secret_key (or SECRET_KEY):

```
$ python -c 'import os; print(os.urandom(16))'
b'_5#y2L"F4Q8z\n\xec]/'
```

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

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

1.4.11 Message Flashing

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

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

1.4.12 Logging

New in version 0.3.

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

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

Here are some example log calls:

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

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

Read more on *Application Errors*.

1.4.13 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)
```

1.4.14 Using Flask Extensions

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

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

1.4.15 Deploying to a Web Server

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

1.5 Tutorial

1.5.1 Project Layout

Create a project directory and enter it:

```
mkdir flask-tutorial
cd flask-tutorial
```

Then follow the *installation instructions* to set up a Python virtual environment and install Flask for your project.

The tutorial will assume you're working from the `flask-tutorial` directory from now on. The file names at the top of each code block are relative to this directory.

A Flask application can be as simple as a single file.

Listing 1.1: `hello.py`

```python
from flask import Flask

app = Flask(__name__)


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

However, as a project get bigger, it becomes overwhelming to keep all the code in one file. Python projects use *packages* to organize code into multiple modules that can be imported where needed, and the tutorial will do this as well.

The project directory will contain:

- `flaskr/`, a Python package containing your application code and files.

- `tests/`, a directory containing test modules.

- `venv/`, a Python virtual environment where Flask and other dependencies are installed.

- Installation files telling Python how to install your project.

- Version control config, such as git. You should make a habit of using some type of version control for all your projects, no matter the size.

- Any other project files you might add in the future.

By the end, your project layout will look like this:

```
/home/user/Projects/flask-tutorial
├── flaskr/
```

```
    ├── __init__.py
    ├── db.py
    ├── schema.sql
    ├── auth.py
    ├── blog.py
    ├── templates/
    │   ├── base.html
    │   ├── auth/
    │   │   ├── login.html
    │   │   └── register.html
    │   └── blog/
    │       ├── create.html
    │       ├── index.html
    │       └── update.html
    └── static/
        └── style.css
├── tests/
│   ├── conftest.py
│   ├── data.sql
│   ├── test_factory.py
│   ├── test_db.py
│   ├── test_auth.py
│   └── test_blog.py
├── venv/
├── setup.py
└── MANIFEST.in
```
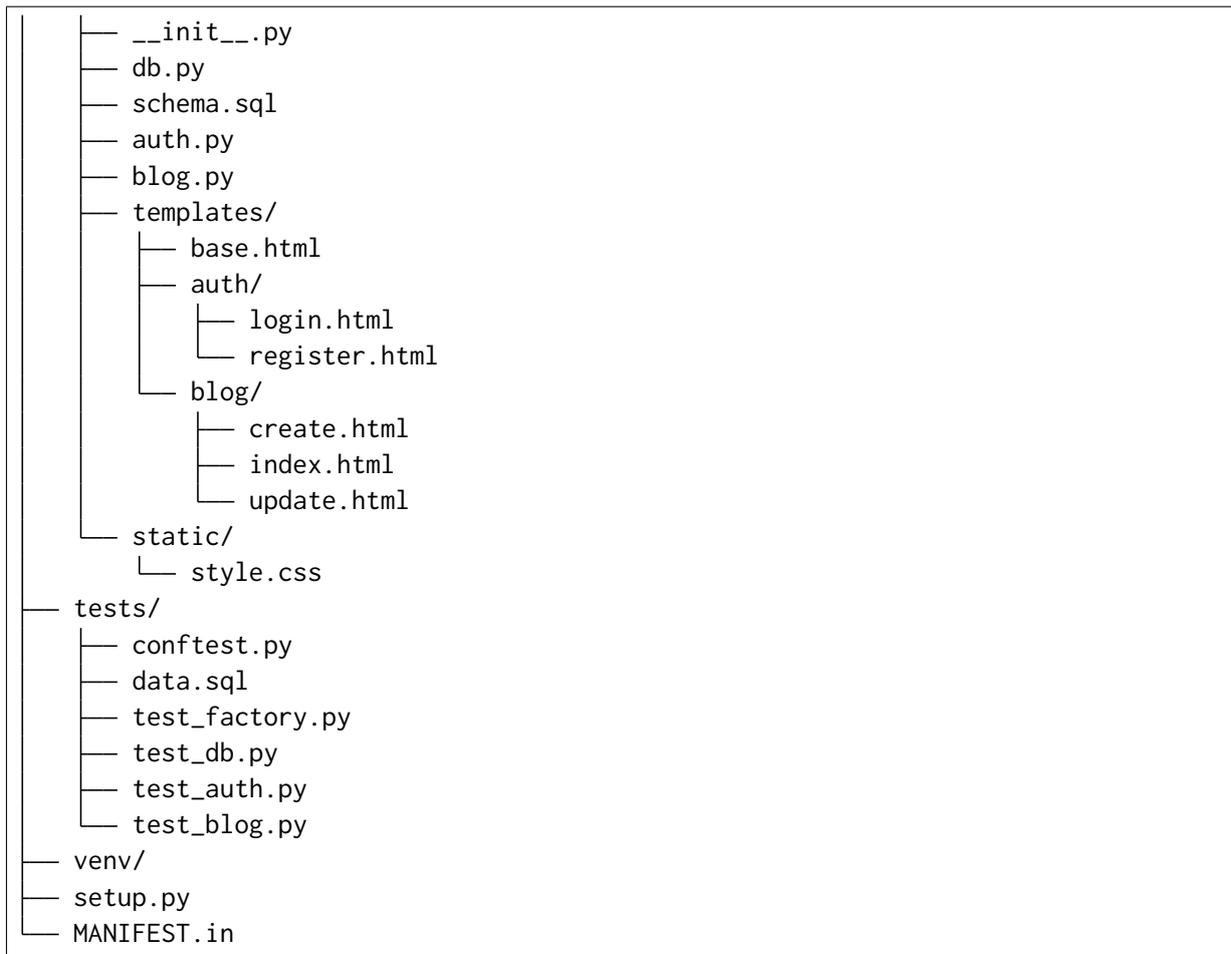
If you're using version control, the following files that are generated while running your project should be ignored. There may be other files based on the editor you use. In general, ignore files that you didn't write. For example, with git:

Listing 1.2: `.gitignore`

```
venv/

*.pyc
__pycache__/

instance/

.pytest_cache/
.coverage
htmlcov/

dist/
build/
*.egg-info/
```

Continue to *Application Setup*.

1.5.2 Application Setup

A Flask application is an instance of the `Flask` class. Everything about the application, such as configuration and URLs, will be registered with this class.

The most straightforward way to create a Flask application is to create a global `Flask` instance directly at the top of your code, like how the "Hello, World!" example did on the previous page. While this is simple and useful in some cases, it can cause some tricky issues as the project grows.

Instead of creating a `Flask` instance globally, you will create it inside a function. This function is known as the *application factory*. Any configuration, registration, and other setup the application needs will happen inside the function, then the application will be returned.

The Application Factory

It's time to start coding! Create the `flaskr` directory and add the `__init__.py` file. The `__init__.py` serves double duty: it will contain the application factory, and it tells Python that the `flaskr` directory should be treated as a package.

```
mkdir flaskr
```

Listing 1.3: `flaskr/__init__.py`

```python
import os

from flask import Flask
```

```python
def create_app(test_config=None):
    # create and configure the app
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_mapping(
        SECRET_KEY='dev',
        DATABASE=os.path.join(app.instance_path, 'flaskr.sqlite'),
    )

    if test_config is None:
        # load the instance config, if it exists, when not testing
        app.config.from_pyfile('config.py', silent=True)
    else:
        # load the test config if passed in
        app.config.from_mapping(test_config)

    # ensure the instance folder exists
    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    # a simple page that says hello
    @app.route('/hello')
    def hello():
        return 'Hello, World!'

    return app
```

create_app is the application factory function. You'll add to it later in the tutorial, but it already does a lot.

1. app = Flask(__name__, instance_relative_config=True) creates the Flask instance.

   - __name__ is the name of the current Python module. The app needs to know where it's located to set up some paths, and __name__ is a convenient way to tell it that.

   - instance_relative_config=True tells the app that configuration files are relative to the *instance folder*. The instance folder is located outside the flaskr package and can hold local data that shouldn't be committed to version control, such as configuration secrets and the database file.

2. app.config.from_mapping() sets some default configuration that the app will use:

   - SECRET_KEY is used by Flask and extensions to keep data safe. It's set to 'dev' to provide a convenient value during development, but it should be overridden with a random value when deploying.

   - DATABASE is the path where the SQLite database file will be saved. It's under app.instance_path, which is the path that Flask has chosen for the instance

---

1.5. Tutorial   25

folder. You'll learn more about the database in the next section.

3. `app.config.from_pyfile()` overrides the default configuration with values taken from the `config.py` file in the instance folder if it exists. For example, when deploying, this can be used to set a real `SECRET_KEY`.

   - `test_config` can also be passed to the factory, and will be used instead of the instance configuration. This is so the tests you'll write later in the tutorial can be configured independently of any development values you have configured.

4. `os.makedirs()` ensures that `app.instance_path` exists. Flask doesn't create the instance folder automatically, but it needs to be created because your project will create the SQLite database file there.

5. `@app.route()` creates a simple route so you can see the application working before getting into the rest of the tutorial. It creates a connection between the URL `/hello` and a function that returns a response, the string `'Hello, World!'` in this case.

Run The Application

Now you can run your application using the `flask` command. From the terminal, tell Flask where to find your application, then run it in development mode.

Development mode shows an interactive debugger whenever a page raises an exception, and restarts the server whenever you make changes to the code. You can leave it running and just reload the browser page as you follow the tutorial.

For Linux and Mac:

```
export FLASK_APP=flaskr
export FLASK_ENV=development
flask run
```

For Windows cmd, use `set` instead of `export`:

```
set FLASK_APP=flaskr
set FLASK_ENV=development
flask run
```

For Windows PowerShell, use `$env:` instead of `export`:

```
$env:FLASK_APP = "flaskr"
$env:FLASK_ENV = "development"
flask run
```

You'll see output similar to this:

```
* Serving Flask app "flaskr"
* Environment: development
* Debug mode: on
```

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 855-212-761
```

Visit http://127.0.0.1:5000/hello in a browser and you should see the "Hello, World!" message. Congratulations, you're now running your Flask web application!

Continue to *Define and Access the Database*.

1.5.3 Define and Access the Database

The application will use a SQLite database to store users and posts. Python comes with built-in support for SQLite in the `sqlite3` module.

SQLite is convenient because it doesn't require setting up a separate database server and is built-in to Python. However, if concurrent requests try to write to the database at the same time, they will slow down as each write happens sequentially. Small applications won't notice this. Once you become big, you may want to switch to a different database.

The tutorial doesn't go into detail about SQL. If you are not familiar with it, the SQLite docs describe the language.

Connect to the Database

The first thing to do when working with a SQLite database (and most other Python database libraries) is to create a connection to it. Any queries and operations are performed using the connection, which is closed after the work is finished.

In web applications this connection is typically tied to the request. It is created at some point when handling a request, and closed before the response is sent.

Listing 1.4: `flaskr/db.py`

```python
import sqlite3

import click
from flask import current_app, g
from flask.cli import with_appcontext


def get_db():
    if 'db' not in g:
        g.db = sqlite3.connect(
            current_app.config['DATABASE'],
            detect_types=sqlite3.PARSE_DECLTYPES
        )
        g.db.row_factory = sqlite3.Row
```

```
    return g.db


def close_db(e=None):
    db = g.pop('db', None)

    if db is not None:
        db.close()
```

g is a special object that is unique for each request. It is used to store data that might
be accessed by multiple functions during the request. The connection is stored and
reused instead of creating a new connection if get_db is called a second time in the
same request.

current_app is another special object that points to the Flask application handling the
request. Since you used an application factory, there is no application object when
writing the rest of your code. get_db will be called when the application has been
created and is handling a request, so current_app can be used.

sqlite3.connect() establishes a connection to the file pointed at by the DATABASE con-
figuration key. This file doesn't have to exist yet, and won't until you initialize the
database later.

sqlite3.Row tells the connection to return rows that behave like dicts. This allows
accessing the columns by name.

close_db checks if a connection was created by checking if g.db was set. If the connec-
tion exists, it is closed. Further down you will tell your application about the close_db
function in the application factory so that it is called after each request.

Create the Tables

In SQLite, data is stored in *tables* and *columns*. These need to be created before you can
store and retrieve data. Flaskr will store users in the user table, and posts in the post
table. Create a file with the SQL commands needed to create empty tables:

Listing 1.5: `flaskr/schema.sql`

```sql
DROP TABLE IF EXISTS user;
DROP TABLE IF EXISTS post;

CREATE TABLE user (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT UNIQUE NOT NULL,
  password TEXT NOT NULL
);

CREATE TABLE post (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  author_id INTEGER NOT NULL,
```

```
  created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  title TEXT NOT NULL,
  body TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES user (id)
);
```

Add the Python functions that will run these SQL commands to the db.py file:

Listing 1.6: `flaskr/db.py`

```python
def init_db():
    db = get_db()

    with current_app.open_resource('schema.sql') as f:
        db.executescript(f.read().decode('utf8'))


@click.command('init-db')
@with_appcontext
def init_db_command():
    """Clear the existing data and create new tables."""
    init_db()
    click.echo('Initialized the database.')
```

open_resource() opens a file relative to the `flaskr` package, which is useful since you won't necessarily know where that location is when deploying the application later. get_db returns a database connection, which is used to execute the commands read from the file.

click.command() defines a command line command called `init-db` that calls the init_db function and shows a success message to the user. You can read *Command Line Interface* to learn more about writing commands.

Register with the Application

The close_db and init_db_command functions need to be registered with the application instance, otherwise they won't be used by the application. However, since you're using a factory function, that instance isn't available when writing the functions. Instead, write a function that takes an application and does the registration.

Listing 1.7: `flaskr/db.py`

```
def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)
```

`app.teardown_appcontext()` tells Flask to call that function when cleaning up after returning the response.

`app.cli.add_command()` adds a new command that can be called with the `flask` command.

Import and call this function from the factory. Place the new code at the end of the factory function before returning the app.

Listing 1.8: `flaskr/__init__.py`

```
def create_app():
    app = ...
    # existing code omitted

    from . import db
    db.init_app(app)

    return app
```

Initialize the Database File

Now that `init-db` has been registered with the app, it can be called using the `flask` command, similar to the `run` command from the previous page.

Note: If you're still running the server from the previous page, you can either stop the server, or run this command in a new terminal. If you use a new terminal, remember to change to your project directory and activate the env as described in *Activate the environment*. You'll also need to set `FLASK_APP` and `FLASK_ENV` as shown on the previous page.

Run the `init-db` command:

```
flask init-db
Initialized the database.
```

There will now be a `flaskr.sqlite` file in the `instance` folder in your project.

Continue to *Blueprints and Views*.

1.5.4 Blueprints and Views

A view function is the code you write to respond to requests to your application. Flask uses patterns to match the incoming request URL to the view that should handle it. The view returns data that Flask turns into an outgoing response. Flask can also go the other direction and generate a URL to a view based on its name and arguments.

Create a Blueprint

A `Blueprint` is a way to organize a group of related views and other code. Rather than registering views and other code directly with an application, they are registered with a blueprint. Then the blueprint is registered with the application when it is available in the factory function.

Flaskr will have two blueprints, one for authentication functions and one for the blog posts functions. The code for each blueprint will go in a separate module. Since the blog needs to know about authentication, you'll write the authentication one first.

Listing 1.9: `flaskr/auth.py`

```python
import functools

from flask import (
    Blueprint, flash, g, redirect, render_template, request, session, url_for
)
from werkzeug.security import check_password_hash, generate_password_hash

from flaskr.db import get_db

bp = Blueprint('auth', __name__, url_prefix='/auth')
```

This creates a `Blueprint` named `'auth'`. Like the application object, the blueprint needs to know where it's defined, so `__name__` is passed as the second argument. The `url_prefix` will be prepended to all the URLs associated with the blueprint.

Import and register the blueprint from the factory using `app.register_blueprint()`. Place the new code at the end of the factory function before returning the app.

Listing 1.10: `flaskr/__init__.py`

```python
def create_app():
    app = ...
    # existing code omitted

    from . import auth
    app.register_blueprint(auth.bp)

    return app
```

The authentication blueprint will have views to register new users and to log in and log out.

The First View: Register

When the user visits the /auth/register URL, the register view will return HTML with a form for them to fill out. When they submit the form, it will validate their input and either show the form again with an error message or create the new user and go to the login page.

For now you will just write the view code. On the next page, you'll write templates to generate the HTML form.

Listing 1.11: `flaskr/auth.py`

```python
@bp.route('/register', methods=('GET', 'POST'))
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None

        if not username:
            error = 'Username is required.'
        elif not password:
            error = 'Password is required.'
        elif db.execute(
            'SELECT id FROM user WHERE username = ?', (username,)
        ).fetchone() is not None:
            error = 'User {} is already registered.'.format(username)

        if error is None:
            db.execute(
                'INSERT INTO user (username, password) VALUES (?, ?)',
                (username, generate_password_hash(password))
            )
            db.commit()
            return redirect(url_for('auth.login'))

        flash(error)

    return render_template('auth/register.html')
```

Here's what the register view function is doing:

1. @bp.route associates the URL /register with the register view function. When Flask receives a request to /auth/register, it will call the register view and use the return value as the response.

2. If the user submitted the form, request.method will be 'POST'. In this case, start validating the input.

3. request.form is a special type of dict mapping submitted form keys and values. The user will input their username and password.

4. Validate that `username` and `password` are not empty.

5. Validate that `username` is not already registered by querying the database and checking if a result is returned. `db.execute` takes a SQL query with ? placeholders for any user input, and a tuple of values to replace the placeholders with. The database library will take care of escaping the values so you are not vulnerable to a *SQL injection attack*.

   `fetchone()` returns one row from the query. If the query returned no results, it returns None. Later, `fetchall()` is used, which returns a list of all results.

6. If validation succeeds, insert the new user data into the database. For security, passwords should never be stored in the database directly. Instead, `generate_password_hash()` is used to securely hash the password, and that hash is stored. Since this query modifies data, `db.commit()` needs to be called afterwards to save the changes.

7. After storing the user, they are redirected to the login page. `url_for()` generates the URL for the login view based on its name. This is preferable to writing the URL directly as it allows you to change the URL later without changing all code that links to it. `redirect()` generates a redirect response to the generated URL.

8. If validation fails, the error is shown to the user. `flash()` stores messages that can be retrieved when rendering the template.

9. When the user initially navigates to `auth/register`, or there was an validation error, an HTML page with the registration form should be shown. `render_template()` will render a template containing the HTML, which you'll write in the next step of the tutorial.

Login

This view follows the same pattern as the `register` view above.

Listing 1.12: `flaskr/auth.py`

```python
@bp.route('/login', methods=('GET', 'POST'))
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None
        user = db.execute(
            'SELECT * FROM user WHERE username = ?', (username,)
        ).fetchone()

        if user is None:
            error = 'Incorrect username.'
        elif not check_password_hash(user['password'], password):
            error = 'Incorrect password.'
```

```
        if error is None:
            session.clear()
            session['user_id'] = user['id']
            return redirect(url_for('index'))

        flash(error)

    return render_template('auth/login.html')
```

There are a few differences from the register view:

1. The user is queried first and stored in a variable for later use.

2. check_password_hash() hashes the submitted password in the same way as the stored hash and securely compares them. If they match, the password is valid.

3. session is a dict that stores data across requests. When validation succeeds, the user's id is stored in a new session. The data is stored in a *cookie* that is sent to the browser, and the browser then sends it back with subsequent requests. Flask securely *signs* the data so that it can't be tampered with.

Now that the user's id is stored in the session, it will be available on subsequent requests. At the beginning of each request, if a user is logged in their information should be loaded and made available to other views.

Listing 1.13: flaskr/auth.py

```
@bp.before_app_request
def load_logged_in_user():
    user_id = session.get('user_id')

    if user_id is None:
        g.user = None
    else:
        g.user = get_db().execute(
            'SELECT * FROM user WHERE id = ?', (user_id,)
        ).fetchone()
```

bp.before_app_request() registers a function that runs before the view function, no matter what URL is requested. load_logged_in_user checks if a user id is stored in the session and gets that user's data from the database, storing it on g.user, which lasts for the length of the request. If there is no user id, or if the id doesn't exist, g.user will be None.

Logout

To log out, you need to remove the user id from the session. Then load_logged_in_user won't load a user on subsequent requests.

Listing 1.14: `flaskr/auth.py`

```python
@bp.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('index'))
```

Require Authentication in Other Views

Creating, editing, and deleting blog posts will require a user to be logged in. A *decorator* can be used to check this for each view it's applied to.

Listing 1.15: `flaskr/auth.py`

```python
def login_required(view):
    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for('auth.login'))

        return view(**kwargs)

    return wrapped_view
```

This decorator returns a new view function that wraps the original view it's applied to. The new function checks if a user is loaded and redirects to the login page otherwise. If a user is loaded the original view is called and continues normally. You'll use this decorator when writing the blog views.

Endpoints and URLs

The `url_for()` function generates the URL to a view based on a name and arguments. The name associated with a view is also called the *endpoint*, and by default it's the same as the name of the view function.

For example, the `hello()` view that was added to the app factory earlier in the tutorial has the name `'hello'` and can be linked to with `url_for('hello')`. If it took an argument, which you'll see later, it would be linked to using `url_for('hello', who='World')`.

When using a blueprint, the name of the blueprint is prepended to the name of the function, so the endpoint for the `login` function you wrote above is `'auth.login'` because you added it to the `'auth'` blueprint.

Continue to *Templates*.

1.5.5 Templates

You've written the authentication views for your application, but if you're running the server and try to go to any of the URLs, you'll see a `TemplateNotFound` error. That's because the views are calling `render_template()`, but you haven't written the templates yet. The template files will be stored in the `templates` directory inside the `flaskr` package.

Templates are files that contain static data as well as placeholders for dynamic data. A template is rendered with specific data to produce a final document. Flask uses the Jinja template library to render templates.

In your application, you will use templates to render HTML which will display in the user's browser. In Flask, Jinja is configured to *autoescape* any data that is rendered in HTML templates. This means that it's safe to render user input; any characters they've entered that could mess with the HTML, such as < and > will be *escaped* with *safe* values that look the same in the browser but don't cause unwanted effects.

Jinja looks and behaves mostly like Python. Special delimiters are used to distinguish Jinja syntax from the static data in the template. Anything between {{ and }} is an expression that will be output to the final document. {% and %} denotes a control flow statement like `if` and `for`. Unlike Python, blocks are denoted by start and end tags rather than indentation since static text within a block could change indentation.

The Base Layout

Each page in the application will have the same basic layout around a different body. Instead of writing the entire HTML structure in each template, each template will *extend* a base template and override specific sections.

Listing 1.16: `flaskr/templates/base.html`

```html
<!doctype html>
<title>{% block title %}{% endblock %} - Flaskr</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
  <h1>Flaskr</h1>
  <ul>
    {% if g.user %}
      <li><span>{{ g.user['username'] }}</span>
      <li><a href="{{ url_for('auth.logout') }}">Log Out</a>
    {% else %}
      <li><a href="{{ url_for('auth.register') }}">Register</a>
      <li><a href="{{ url_for('auth.login') }}">Log In</a>
    {% endif %}
  </ul>
</nav>
<section class="content">
  <header>
    {% block header %}{% endblock %}
```

```
    </header>
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  {% block content %}{% endblock %}
</section>
```

`g` is automatically available in templates. Based on if `g.user` is set (from `load_logged_in_user`), either the username and a log out link are displayed, otherwise links to register and log in are displayed. `url_for()` is also automatically available, and is used to generate URLs to views instead of writing them out manually.

After the page title, and before the content, the template loops over each message returned by `get_flashed_messages()`. You used `flash()` in the views to show error messages, and this is the code that will display them.

There are three blocks defined here that will be overridden in the other templates:

1. `{% block title %}` will change the title displayed in the browser's tab and window title.

2. `{% block header %}` is similar to `title` but will change the title displayed on the page.

3. `{% block content %}` is where the content of each page goes, such as the login form or a blog post.

The base template is directly in the `templates` directory. To keep the others organized, the templates for a blueprint will be placed in a directory with the same name as the blueprint.

Register

Listing 1.17: `flaskr/templates/auth/register.html`

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Register{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Register">
  </form>
{% endblock %}
```

{% extends 'base.html' %} tells Jinja that this template should replace the blocks from the base template. All the rendered content must appear inside {% block %} tags that override blocks from the base template.

A useful pattern used here is to place {% block title %} inside {% block header %}. This will set the title block and then output the value of it into the header block, so that both the window and page share the same title without writing it twice.

The input tags are using the required attribute here. This tells the browser not to submit the form until those fields are filled in. If the user is using an older browser that doesn't support that attribute, or if they are using something besides a browser to make requests, you still want to validate the data in the Flask view. It's important to always fully validate the data on the server, even if the client does some validation as well.

Log In

This is identical to the register template except for the title and submit button.

Listing 1.18: flaskr/templates/auth/login.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Log In{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Log In">
  </form>
{% endblock %}
```

Register A User

Now that the authentication templates are written, you can register a user. Make sure the server is still running (flask run if it's not), then go to http://127.0.0.1:5000/auth/register.

Try clicking the "Register" button without filling out the form and see that the browser shows an error message. Try removing the required attributes from the register.html template and click "Register" again. Instead of the browser showing an error, the page will reload and the error from flash() in the view will be shown.

Fill out a username and password and you'll be redirected to the login page. Try entering an incorrect username, or the correct username and incorrect password. If

you log in you'll get an error because there's no `index` view to redirect to yet.

Continue to *Static Files*.

1.5.6 Static Files

The authentication views and templates work, but they look very plain right now. Some CSS can be added to add style to the HTML layout you constructed. The style won't change, so it's a *static* file rather than a template.

Flask automatically adds a `static` view that takes a path relative to the `flaskr/static` directory and serves it. The `base.html` template already has a link to the `style.css` file:

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

Besides CSS, other types of static files might be files with JavaScript functions, or a logo image. They are all placed under the `flaskr/static` directory and referenced with `url_for('static', filename='...')`.

This tutorial isn't focused on how to write CSS, so you can just copy the following into the `flaskr/static/style.css` file:

Listing 1.19: `flaskr/static/style.css`

```css
html { font-family: sans-serif; background: #eee; padding: 1rem; }
body { max-width: 960px; margin: 0 auto; background: white; }
h1 { font-family: serif; color: #377ba8; margin: 1rem 0; }
a { color: #377ba8; }
hr { border: none; border-top: 1px solid lightgray; }
nav { background: lightgray; display: flex; align-items: center; padding: 0 0.
↪5rem; }
nav h1 { flex: auto; margin: 0; }
nav h1 a { text-decoration: none; padding: 0.25rem 0.5rem; }
nav ul  { display: flex; list-style: none; margin: 0; padding: 0; }
nav ul li a, nav ul li span, header .action { display: block; padding: 0.5rem; }
.content { padding: 0 1rem 1rem; }
.content > header { border-bottom: 1px solid lightgray; display: flex; align-
↪items: flex-end; }
.content > header h1 { flex: auto; margin: 1rem 0 0.25rem 0; }
.flash { margin: 1em 0; padding: 1em; background: #cae6f6; border: 1px solid
↪#377ba8; }
.post > header { display: flex; align-items: flex-end; font-size: 0.85em; }
.post > header > div:first-of-type { flex: auto; }
.post > header h1 { font-size: 1.5em; margin-bottom: 0; }
.post .about { color: slategray; font-style: italic; }
.post .body { white-space: pre-line; }
.content:last-child { margin-bottom: 0; }
.content form { margin: 1em 0; display: flex; flex-direction: column; }
.content label { font-weight: bold; margin-bottom: 0.5em; }
.content input, .content textarea { margin-bottom: 1em; }
```

```
.content textarea { min-height: 12em; resize: vertical; }
input.danger { color: #cc2f2e; }
input[type=submit] { align-self: start; min-width: 10em; }
```

You can find a less compact version of `style.css` in the example code.

Go to http://127.0.0.1/auth/login and the page should look like the screenshot below.

You can read more about CSS from Mozilla's documentation. If you change a static file, refresh the browser page. If the change doesn't show up, try clearing your browser's cache.

Continue to *Blog Blueprint*.

1.5.7 Blog Blueprint

You'll use the same techniques you learned about when writing the authentication blueprint to write the blog blueprint. The blog should list all posts, allow logged in users to create posts, and allow the author of a post to edit or delete it.

As you implement each view, keep the development server running. As you save your changes, try going to the URL in your browser and testing them out.

The Blueprint

Define the blueprint and register it in the application factory.

Listing 1.20: `flaskr/blog.py`

```python
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort

from flaskr.auth import login_required
from flaskr.db import get_db

bp = Blueprint('blog', __name__)
```

Import and register the blueprint from the factory using `app.register_blueprint()`.
Place the new code at the end of the factory function before returning the app.

Listing 1.21: `flaskr/__init__.py`

```python
def create_app():
    app = ...
    # existing code omitted

    from . import blog
    app.register_blueprint(blog.bp)
    app.add_url_rule('/', endpoint='index')

    return app
```

Unlike the auth blueprint, the blog blueprint does not have a `url_prefix`. So the `index`
view will be at `/`, the `create` view at `/create`, and so on. The blog is the main feature
of Flaskr, so it makes sense that the blog index will be the main index.

However, the endpoint for the `index` view defined below will be `blog.index`. Some
of the authentication views referred to a plain `index` endpoint. `app.add_url_rule()`
associates the endpoint name `'index'` with the `/` url so that `url_for('index')` or
`url_for('blog.index')` will both work, generating the same `/` URL either way.

In another application you might give the blog blueprint a `url_prefix` and define a
separate `index` view in the application factory, similar to the `hello` view. Then the
`index` and `blog.index` endpoints and URLs would be different.

Index

The index will show all of the posts, most recent first. A `JOIN` is used so that the author
information from the user table is available in the result.

Listing 1.22: `flaskr/blog.py`

```python
@bp.route('/')
def index():
    db = get_db()
    posts = db.execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM post p JOIN user u ON p.author_id = u.id'
        ' ORDER BY created DESC'
    ).fetchall()
    return render_template('blog/index.html', posts=posts)
```

Listing 1.23: `flaskr/templates/blog/index.html`

```html
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Posts{% endblock %}</h1>
  {% if g.user %}
    <a class="action" href="{{ url_for('blog.create') }}">New</a>
  {% endif %}
{% endblock %}

{% block content %}
  {% for post in posts %}
    <article class="post">
      <header>
        <div>
          <h1>{{ post['title'] }}</h1>
          <div class="about">by {{ post['username'] }} on {{ post['created'].
→strftime('%Y-%m-%d') }}</div>
        </div>
        {% if g.user['id'] == post['author_id'] %}
          <a class="action" href="{{ url_for('blog.update', id=post['id']) }}">
→Edit</a>
        {% endif %}
      </header>
      <p class="body">{{ post['body'] }}</p>
    </article>
    {% if not loop.last %}
      <hr>
    {% endif %}
  {% endfor %}
{% endblock %}
```

When a user is logged in, the `header` block adds a link to the `create` view. When the user is the author of a post, they'll see an "Edit" link to the `update` view for that post. `loop.last` is a special variable available inside Jinja for loops. It's used to display a line after each post except the last one, to visually separate them.

Create

The `create` view works the same as the auth `register` view. Either the form is displayed, or the posted data is validated and the post is added to the database or an error is shown.

The `login_required` decorator you wrote earlier is used on the blog views. A user must be logged in to visit these views, otherwise they will be redirected to the login page.

Listing 1.24: `flaskr/blog.py`

```python
@bp.route('/create', methods=('GET', 'POST'))
@login_required
def create():
    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'INSERT INTO post (title, body, author_id)'
                ' VALUES (?, ?, ?)',
                (title, body, g.user['id'])
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/create.html')
```

Listing 1.25: `flaskr/templates/blog/create.html`

```html
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}New Post{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="title">Title</label>
    <input name="title" id="title" value="{{ request.form['title'] }}" required>
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] }}</textarea>
```

```
    <input type="submit" value="Save">
  </form>
{% endblock %}
```

Update

Both the `update` and `delete` views will need to fetch a `post` by `id` and check if the author matches the logged in user. To avoid duplicating code, you can write a function to get the `post` and call it from each view.

Listing 1.26: `flaskr/blog.py`

```python
def get_post(id, check_author=True):
    post = get_db().execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM post p JOIN user u ON p.author_id = u.id'
        ' WHERE p.id = ?',
        (id,)
    ).fetchone()

    if post is None:
        abort(404, "Post id {0} doesn't exist.".format(id))

    if check_author and post['author_id'] != g.user['id']:
        abort(403)

    return post
```

`abort()` will raise a special exception that returns an HTTP status code. It takes an optional message to show with the error, otherwise a default message is used. `404` means "Not Found", and `403` means "Forbidden". (`401` means "Unauthorized", but you redirect to the login page instead of returning that status.)

The `check_author` argument is defined so that the function can be used to get a `post` without checking the author. This would be useful if you wrote a view to show an individual post on a page, where the user doesn't matter because they're not modifying the post.

Listing 1.27: `flaskr/blog.py`

```python
@bp.route('/<int:id>/update', methods=('GET', 'POST'))
@login_required
def update(id):
    post = get_post(id)

    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None
```

```
        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'UPDATE post SET title = ?, body = ?'
                ' WHERE id = ?',
                (title, body, id)
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/update.html', post=post)
```

Unlike the views you've written so far, the update function takes an argument, id.
That corresponds to the <int:id> in the route. A real URL will look like /1/update.
Flask will capture the 1, ensure it's an int, and pass it as the id argument. If you don't
specify int: and instead do <id>, it will be a string. To generate a URL to the update
page, url_for() needs to be passed the id so it knows what to fill in: url_for('blog.
update', id=post['id']). This is also in the index.html file above.

The create and update views look very similar. The main difference is that the update
view uses a post object and an UPDATE query instead of an INSERT. With some clever
refactoring, you could use one view and template for both actions, but for the tutorial
it's clearer to keep them separate.

Listing 1.28: flaskr/templates/blog/update.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Edit "{{ post['title'] }}"{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="title">Title</label>
    <input name="title" id="title"
      value="{{ request.form['title'] or post['title'] }}" required>
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] or post['body'] }}</
→textarea>
    <input type="submit" value="Save">
  </form>
  <hr>
  <form action="{{ url_for('blog.delete', id=post['id']) }}" method="post">
    <input class="danger" type="submit" value="Delete" onclick="return confirm(
→'Are you sure?');">
```

```
    </form>
{% endblock %}
```

This template has two forms. The first posts the edited data to the current page (/
<id>/update). The other form contains only a button and specifies an action attribute
that posts to the delete view instead. The button uses some JavaScript to show a con-
firmation dialog before submitting.

The pattern {{ request.form['title'] or post['title'] }} is used to choose what
data appears in the form. When the form hasn't been submitted, the original post
data appears, but if invalid form data was posted you want to display that so the user
can fix the error, so request.form is used instead. request is another variable that's
automatically available in templates.

Delete

The delete view doesn't have its own template, the delete button is part of update.html
and posts to the /<id>/delete URL. Since there is no template, it will only handle the
POST method then redirect to the index view.

Listing 1.29: flaskr/blog.py

```python
@bp.route('/<int:id>/delete', methods=('POST',))
@login_required
def delete(id):
    get_post(id)
    db = get_db()
    db.execute('DELETE FROM post WHERE id = ?', (id,))
    db.commit()
    return redirect(url_for('blog.index'))
```

Congratulations, you've now finished writing your application! Take some time to try
out everything in the browser. However, there's still more to do before the project is
complete.

Continue to *Make the Project Installable*.

1.5.8 Make the Project Installable

Making your project installable means that you can build a *distribution* file and install
that in another environment, just like you installed Flask in your project's environ-
ment. This makes deploying your project the same as installing any other library, so
you're using all the standard Python tools to manage everything.

Installing also comes with other benefits that might not be obvious from the tutorial
or as a new Python user, including:

- Currently, Python and Flask understand how to use the flaskr package only
  because you're running from your project's directory. Installing means you can

import it no matter where you run from.

- You can manage your project's dependencies just like other packages do, so `pip install yourproject.whl` installs them.

- Test tools can isolate your test environment from your development environment.

Note: This is being introduced late in the tutorial, but in your future projects you should always start with this.

Describe the Project

The `setup.py` file describes your project and the files that belong to it.

Listing 1.30: `setup.py`

```python
from setuptools import find_packages, setup

setup(
    name='flaskr',
    version='1.0.0',
    packages=find_packages(),
    include_package_data=True,
    zip_safe=False,
    install_requires=[
        'flask',
    ],
)
```

packages tells Python what package directories (and the Python files they contain) to include. `find_packages()` finds these directories automatically so you don't have to type them out. To include other files, such as the static and templates directories, `include_package_data` is set. Python needs another file named `MANIFEST.in` to tell what this other data is.

Listing 1.31: `MANIFEST.in`

```
include flaskr/schema.sql
graft flaskr/static
graft flaskr/templates
global-exclude *.pyc
```

This tells Python to copy everything in the `static` and `templates` directories, and the `schema.sql` file, but to exclude all bytecode files.

See the official packaging guide for another explanation of the files and options used.

Install the Project

Use `pip` to install your project in the virtual environment.

```
pip install -e .
```

This tells pip to find `setup.py` in the current directory and install it in *editable* or *development* mode. Editable mode means that as you make changes to your local code, you'll only need to re-install if you change the metadata about the project, such as its dependencies.

You can observe that the project is now installed with `pip list`.

```
pip list

Package        Version   Location
-------------- --------- --------------------------------
click          6.7
Flask          1.0
flaskr         1.0.0     /home/user/Projects/flask-tutorial
itsdangerous   0.24
Jinja2         2.10
MarkupSafe     1.0
pip            9.0.3
setuptools     39.0.1
Werkzeug       0.14.1
wheel          0.30.0
```

Nothing changes from how you've been running your project so far. `FLASK_APP` is still set to `flaskr` and `flask run` still runs the application.

Continue to *Test Coverage*.

1.5.9 Test Coverage

Writing unit tests for your application lets you check that the code you wrote works the way you expect. Flask provides a test client that simulates requests to the application and returns the response data.

You should test as much of your code as possible. Code in functions only runs when the function is called, and code in branches, such as `if` blocks, only runs when the condition is met. You want to make sure that each function is tested with data that covers each branch.

The closer you get to 100% coverage, the more comfortable you can be that making a change won't unexpectedly change other behavior. However, 100% coverage doesn't guarantee that your application doesn't have bugs. In particular, it doesn't test how the user interacts with the application in the browser. Despite this, test coverage is an important tool to use during development.

Note: This is being introduced late in the tutorial, but in your future projects you should test as you develop.

You'll use pytest and coverage to test and measure your code. Install them both:

```
pip install pytest coverage
```

Setup and Fixtures

The test code is located in the tests directory. This directory is *next to* the flaskr package, not inside it. The tests/conftest.py file contains setup functions called *fixtures* that each test will use. Tests are in Python modules that start with test_, and each test function in those modules also starts with test_.

Each test will create a new temporary database file and populate some data that will be used in the tests. Write a SQL file to insert that data.

Listing 1.32: `tests/data.sql`

```sql
INSERT INTO user (username, password)
VALUES
  ('test', 'pbkdf2:sha256:50000$TCI4GzcX
↪$0de171a4f4dac32e3364c7ddc7c14f3e2fa61f2d17574483f7ffbb431b4acb2f'),
  ('other', 'pbkdf2:sha256:50000$kJPKsz6N
↪$d2d4784f1b030a9761f5ccaeeaca413f27f2ecb76d6168407af962ddce849f79');


INSERT INTO post (title, body, author_id, created)
VALUES
  ('test title', 'test' || x'0a' || 'body', 1, '2018-01-01 00:00:00');
```

The app fixture will call the factory and pass test_config to configure the application and database for testing instead of using your local development configuration.

Listing 1.33: `tests/conftest.py`

```python
import os
import tempfile

import pytest
from flaskr import create_app
from flaskr.db import get_db, init_db

with open(os.path.join(os.path.dirname(__file__), 'data.sql'), 'rb') as f:
    _data_sql = f.read().decode('utf8')


@pytest.fixture
def app():
    db_fd, db_path = tempfile.mkstemp()
```

```
    app = create_app({
        'TESTING': True,
        'DATABASE': db_path,
    })

    with app.app_context():
        init_db()
        get_db().executescript(_data_sql)

    yield app

    os.close(db_fd)
    os.unlink(db_path)


@pytest.fixture
def client(app):
    return app.test_client()


@pytest.fixture
def runner(app):
    return app.test_cli_runner()
```

`tempfile.mkstemp()` creates and opens a temporary file, returning the file object and the path to it. The DATABASE path is overridden so it points to this temporary path instead of the instance folder. After setting the path, the database tables are created and the test data is inserted. After the test is over, the temporary file is closed and removed.

TESTING tells Flask that the app is in test mode. Flask changes some internal behavior so it's easier to test, and other extensions can also use the flag to make testing them easier.

The `client` fixture calls `app.test_client()` with the application object created by the app fixture. Tests will use the client to make requests to the application without running the server.

The `runner` fixture is similar to `client`. `app.test_cli_runner()` creates a runner that can call the Click commands registered with the application.

Pytest uses fixtures by matching their function names with the names of arguments in the test functions. For example, the `test_hello` function you'll write next takes a `client` argument. Pytest matches that with the `client` fixture function, calls it, and passes the returned value to the test function.

Factory

There's not much to test about the factory itself. Most of the code will be executed for each test already, so if something fails the other tests will notice.

The only behavior that can change is passing test config. If config is not passed, there should be some default configuration, otherwise the configuration should be overridden.

Listing 1.34: `tests/test_factory.py`

```python
from flaskr import create_app


def test_config():
    assert not create_app().testing
    assert create_app({'TESTING': True}).testing


def test_hello(client):
    response = client.get('/hello')
    assert response.data == b'Hello, World!'
```

You added the `hello` route as an example when writing the factory at the beginning of the tutorial. It returns "Hello, World!", so the test checks that the response data matches.

Database

Within an application context, `get_db` should return the same connection each time it's called. After the context, the connection should be closed.

Listing 1.35: `tests/test_db.py`

```python
import sqlite3

import pytest
from flaskr.db import get_db


def test_get_close_db(app):
    with app.app_context():
        db = get_db()
        assert db is get_db()

    with pytest.raises(sqlite3.ProgrammingError) as e:
        db.execute('SELECT 1')

    assert 'closed' in str(e)
```

The `init-db` command should call the `init_db` function and output a message.

Listing 1.36: `tests/test_db.py`

```python
def test_init_db_command(runner, monkeypatch):
    class Recorder(object):
        called = False

    def fake_init_db():
        Recorder.called = True

    monkeypatch.setattr('flaskr.db.init_db', fake_init_db)
    result = runner.invoke(args=['init-db'])
    assert 'Initialized' in result.output
    assert Recorder.called
```

This test uses Pytest's monkeypatch fixture to replace the `init_db` function with one that records that it's been called. The `runner` fixture you wrote above is used to call the `init-db` command by name.

Authentication

For most of the views, a user needs to be logged in. The easiest way to do this in tests is to make a `POST` request to the `login` view with the client. Rather than writing that out every time, you can write a class with methods to do that, and use a fixture to pass it the client for each test.

Listing 1.37: `tests/conftest.py`

```python
class AuthActions(object):
    def __init__(self, client):
        self._client = client

    def login(self, username='test', password='test'):
        return self._client.post(
            '/auth/login',
            data={'username': username, 'password': password}
        )

    def logout(self):
        return self._client.get('/auth/logout')


@pytest.fixture
def auth(client):
    return AuthActions(client)
```

With the auth fixture, you can call `auth.login()` in a test to log in as the `test` user, which was inserted as part of the test data in the app fixture.

The `register` view should render successfully on `GET`. On `POST` with valid form data, it should redirect to the login URL and the user's data should be in the database. Invalid

data should display error messages.

Listing 1.38: tests/test_auth.py

```python
import pytest
from flask import g, session
from flaskr.db import get_db


def test_register(client, app):
    assert client.get('/auth/register').status_code == 200
    response = client.post(
        '/auth/register', data={'username': 'a', 'password': 'a'}
    )
    assert 'http://localhost/auth/login' == response.headers['Location']

    with app.app_context():
        assert get_db().execute(
            "select * from user where username = 'a'",
        ).fetchone() is not None


@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('', '', b'Username is required.'),
    ('a', '', b'Password is required.'),
    ('test', 'test', b'already registered'),
))
def test_register_validate_input(client, username, password, message):
    response = client.post(
        '/auth/register',
        data={'username': username, 'password': password}
    )
    assert message in response.data
```

client.get() makes a GET request and returns the Response object returned by Flask. Similarly, client.post() makes a POST request, converting the data dict into form data.

To test that the page renders successfully, a simple request is made and checked for a 200 OK status_code. If rendering failed, Flask would return a 500 Internal Server Error code.

headers will have a Location header with the login URL when the register view redirects to the login view.

data contains the body of the response as bytes. If you expect a certain value to render on the page, check that it's in data. Bytes must be compared to bytes. If you want to compare Unicode text, use get_data(as_text=True) instead.

pytest.mark.parametrize tells Pytest to run the same test function with different arguments. You use it here to test different invalid input and error messages without writing the same code three times.

The tests for the login view are very similar to those for register. Rather than testing

the data in the database, `session` should have `user_id` set after logging in.

Listing 1.39: `tests/test_auth.py`

```python
def test_login(client, auth):
    assert client.get('/auth/login').status_code == 200
    response = auth.login()
    assert response.headers['Location'] == 'http://localhost/'

    with client:
        client.get('/')
        assert session['user_id'] == 1
        assert g.user['username'] == 'test'


@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('a', 'test', b'Incorrect username.'),
    ('test', 'a', b'Incorrect password.'),
))
def test_login_validate_input(auth, username, password, message):
    response = auth.login(username, password)
    assert message in response.data
```

Using `client` in a `with` block allows accessing context variables such as `session` after the response is returned. Normally, accessing `session` outside of a request would raise an error.

Testing `logout` is the opposite of `login`. `session` should not contain `user_id` after logging out.

Listing 1.40: `tests/test_auth.py`

```python
def test_logout(client, auth):
    auth.login()

    with client:
        auth.logout()
        assert 'user_id' not in session
```

Blog

All the blog views use the auth fixture you wrote earlier. Call `auth.login()` and subsequent requests from the client will be logged in as the `test` user.

The `index` view should display information about the post that was added with the test data. When logged in as the author, there should be a link to edit the post.

You can also test some more authentication behavior while testing the `index` view. When not logged in, each page shows links to log in or register. When logged in, there's a link to log out.

Listing 1.41: tests/test_blog.py

```python
import pytest
from flaskr.db import get_db


def test_index(client, auth):
    response = client.get('/')
    assert b"Log In" in response.data
    assert b"Register" in response.data

    auth.login()
    response = client.get('/')
    assert b'Log Out' in response.data
    assert b'test title' in response.data
    assert b'by test on 2018-01-01' in response.data
    assert b'test\nbody' in response.data
    assert b'href="/1/update"' in response.data
```

A user must be logged in to access the create, update, and delete views. The logged in user must be the author of the post to access update and delete, otherwise a 403 Forbidden status is returned. If a post with the given id doesn't exist, update and delete should return 404 Not Found.

Listing 1.42: tests/test_blog.py

```python
@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
    '/1/delete',
))
def test_login_required(client, path):
    response = client.post(path)
    assert response.headers['Location'] == 'http://localhost/auth/login'


def test_author_required(app, client, auth):
    # change the post author to another user
    with app.app_context():
        db = get_db()
        db.execute('UPDATE post SET author_id = 2 WHERE id = 1')
        db.commit()

    auth.login()
    # current user can't modify other user's post
    assert client.post('/1/update').status_code == 403
    assert client.post('/1/delete').status_code == 403
    # current user doesn't see edit link
    assert b'href="/1/update"' not in client.get('/').data
```

```
@pytest.mark.parametrize('path', (
    '/2/update',
    '/2/delete',
))
def test_exists_required(client, auth, path):
    auth.login()
    assert client.post(path).status_code == 404
```

The create and update views should render and return a 200 OK status for a GET request. When valid data is sent in a POST request, create should insert the new post data into the database, and update should modify the existing data. Both pages should show an error message on invalid data.

Listing 1.43: tests/test_blog.py

```
def test_create(client, auth, app):
    auth.login()
    assert client.get('/create').status_code == 200
    client.post('/create', data={'title': 'created', 'body': ''})

    with app.app_context():
        db = get_db()
        count = db.execute('SELECT COUNT(id) FROM post').fetchone()[0]
        assert count == 2


def test_update(client, auth, app):
    auth.login()
    assert client.get('/1/update').status_code == 200
    client.post('/1/update', data={'title': 'updated', 'body': ''})

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post['title'] == 'updated'


@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
))
def test_create_update_validate(client, auth, path):
    auth.login()
    response = client.post(path, data={'title': '', 'body': ''})
    assert b'Title is required.' in response.data
```

The delete view should redirect to the index URL and the post should no longer exist in the database.

Listing 1.44: `tests/test_blog.py`

```python
def test_delete(client, auth, app):
    auth.login()
    response = client.post('/1/delete')
    assert response.headers['Location'] == 'http://localhost/'

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post is None
```

Running the Tests

Some extra configuration, which is not required but makes running tests with coverage less verbose, can be added to the project's `setup.cfg` file.

Listing 1.45: `setup.cfg`

```
[tool:pytest]
testpaths = tests

[coverage:run]
branch = True
source =
    flaskr
```

To run the tests, use the `pytest` command. It will find and run all the test functions you've written.

```
pytest

========================= test session starts ==========================
platform linux -- Python 3.6.4, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /home/user/Projects/flask-tutorial, inifile: setup.cfg
collected 23 items

tests/test_auth.py ........                                       [ 34%]
tests/test_blog.py ............                                   [ 86%]
tests/test_db.py ..                                               [ 95%]
tests/test_factory.py ..                                          [100%]

===================== 24 passed in 0.64 seconds ========================
```

If any tests fail, pytest will show the error that was raised. You can run `pytest -v` to get a list of each test function rather than dots.

To measure the code coverage of your tests, use the `coverage` command to run pytest instead of running it directly.

```
coverage run -m pytest
```

You can either view a simple coverage report in the terminal:

```
coverage report

Name                  Stmts   Miss Branch BrPart  Cover
--------------------------------------------------------
flaskr/__init__.py       21      0      2      0   100%
flaskr/auth.py           54      0     22      0   100%
flaskr/blog.py           54      0     16      0   100%
flaskr/db.py             24      0      4      0   100%
--------------------------------------------------------
TOTAL                   153      0     44      0   100%
```

An HTML report allows you to see which lines were covered in each file:

```
coverage html
```

This generates files in the `htmlcov` directory. Open `htmlcov/index.html` in your browser to see the report.

Continue to *Deploy to Production*.

1.5.10 Deploy to Production

This part of the tutorial assumes you have a server that you want to deploy your application to. It gives an overview of how to create the distribution file and install it, but won't go into specifics about what server or software to use. You can set up a new environment on your development computer to try out the instructions below, but probably shouldn't use it for hosting a real public application. See *Deployment Options* for a list of many different ways to host your application.

Build and Install

When you want to deploy your application elsewhere, you build a distribution file. The current standard for Python distribution is the *wheel* format, with the `.whl` extension. Make sure the wheel library is installed first:

```
pip install wheel
```

Running `setup.py` with Python gives you a command line tool to issue build-related commands. The `bdist_wheel` command will build a wheel distribution file.

```
python setup.py bdist_wheel
```

You can find the file in `dist/flaskr-1.0.0-py3-none-any.whl`. The file name is the name of the project, the version, and some tags about the file can install.

Copy this file to another machine, *set up a new virtualenv*, then install the file with pip.

```
pip install flaskr-1.0.0-py3-none-any.whl
```

Pip will install your project along with its dependencies.

Since this is a different machine, you need to run `init-db` again to create the database in the instance folder.

```
export FLASK_APP=flaskr
flask init-db
```

When Flask detects that it's installed (not in editable mode), it uses a different directory for the instance folder. You can find it at `venv/var/flaskr-instance` instead.

Configure the Secret Key

In the beginning of the tutorial that you gave a default value for `SECRET_KEY`. This should be changed to some random bytes in production. Otherwise, attackers could use the public `'dev'` key to modify the session cookie, or anything else that uses the secret key.

You can use the following command to output a random secret key:

```
python -c 'import os; print(os.urandom(16))'

b'_5#y2L"F4Q8z\n\xec]/'
```

Create the `config.py` file in the instance folder, which the factory will read from if it exists. Copy the generated value into it.

Listing 1.46: `venv/var/flaskr-instance/config.py`

```
SECRET_KEY = b'_5#y2L"F4Q8z\n\xec]/'
```

You can also set any other necessary configuration here, although `SECRET_KEY` is the only one needed for Flaskr.

Run with a Production Server

When running publicly rather than in development, you should not use the built-in development server (`flask run`). The development server is provided by Werkzeug for convenience, but is not designed to be particularly efficient, stable, or secure.

Instead, use a production WSGI server. For example, to use Waitress, first install it in the virtual environment:

```
pip install waitress
```

You need to tell Waitress about your application, but it doesn't use `FLASK_APP` like `flask run` does. You need to tell it to import and call the application factory to get an application object.

```
waitress-serve --call 'flaskr:create_app'

Serving on http://0.0.0.0:8080
```

See *Deployment Options* for a list of many different ways to host your application. Waitress is just an example, chosen for the tutorial because it supports both Windows and Linux. There are many more WSGI servers and deployment options that you may choose for your project.

Continue to *Keep Developing!*.

1.5.11 Keep Developing!

You've learned about quite a few Flask and Python concepts throughout the tutorial. Go back and review the tutorial and compare your code with the steps you took to get there. Compare your project to the example project, which might look a bit different due to the step-by-step nature of the tutorial.

There's a lot more to Flask than what you've seen so far. Even so, you're now equipped to start developing your own web applications. Check out the *Quickstart* for an overview of what Flask can do, then dive into the docs to keep learning. Flask uses Jinja, Click, Werkzeug, and ItsDangerous behind the scenes, and they all have their own documentation too. You'll also be interested in *Extensions* which make tasks like working with the database or validating form data easier and more powerful.

If you want to keep developing your Flaskr project, here are some ideas for what to try next:

- A detail view to show a single post. Click a post's title to go to its page.
- Like / unlike a post.
- Comments.
- Tags. Clicking a tag shows all the posts with that tag.
- A search box that filters the index page by name.
- Paged display. Only show 5 posts per page.
- Upload an image to go along with a post.
- Format posts using Markdown.
- An RSS feed of new posts.

Have fun and make awesome applications!

This tutorial will walk you through creating a basic blog application called Flaskr. Users will be able to register, log in, create posts, and edit or delete their own posts. You will be able to package and install the application on other computers.

It's assumed that you're already familiar with Python. The official tutorial in the Python docs is a great way to learn or review first.

While it's designed to give a good starting point, the tutorial doesn't cover all of Flask's features. Check out the *Quickstart* for an overview of what Flask can do, then dive into the docs to find out more. The tutorial only uses what's provided by Flask and Python. In another project, you might decide to use *Extensions* or other libraries to make some tasks simpler.

Flask is flexible. It doesn't require you to use any particular project or code layout. However, when first starting, it's helpful to use a more structured approach. This means that the tutorial will require a bit of boilerplate up front, but it's done to avoid many common pitfalls that new developers encounter, and it creates a project that's easy to expand on. Once you become more comfortable with Flask, you can step out of this structure and take full advantage of Flask's flexibility.

The tutorial project is available as an example in the Flask repository, if you want to compare your project with the final product as you follow the tutorial.

Continue to *Project Layout*.

1.6 Templates

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

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

1.6.1 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` when using `render_template()`.

- autoescaping is enabled for all strings when using `render_template_string()`.

- a template has the ability to opt in/out autoescaping with the `{% autoescape %}` tag.

- Flask inserts a couple of global functions and helpers into the Jinja2 context, additionally to the values that are present by default.

1.6.2 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.
>
> Changed in version 0.10: This is now always available, even in imported templates.

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

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

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

url_for()
> The `flask.url_for()` function.

get_flashed_messages()
> The `flask.get_flashed_messages()` function.

The Jinja Context Behavior

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

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

1. you explicitly pass the request to the macro as parameter, or the attribute of the request object you are interested in.

2. you import the macro "with context".

Importing with context looks like this:

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

1.6.3 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` before Flask 0.10 if you intend to use it inside `script` tags:

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

1.6.4 Controlling Autoescaping

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

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

There are three ways to accomplish that:

- In the Python code, wrap the HTML string in a `Markup` object before passing it to the template. This is in general the recommended way.

- Inside the template, use the |safe filter to explicitly mark a string as safe HTML
  ({{ myvariable|safe }})

- Temporarily disable the autoescape system altogether.

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

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

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

1.6.5 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. Once registered, you can use the filter in your templates in the
same way as Jinja2's builtin filters, for example if you have a Python list in context
called *mylist*:

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

1.6.6 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, for all templates in the app:

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

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

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

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

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

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

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

1.7 Testing Flask Applications

> **Something that is untested is broken.**

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

Flask provides a way to test your application by exposing the Werkzeug test `Client` and handling the context locals for you. You can then use that with your favourite testing solution.

In this documentation we will use the pytest package as the base framework for our tests. You can install it with `pip`, like so:

```
pip install pytest
```

1.7.1 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 source code from the examples.

1.7.2 The Testing Skeleton

We begin by adding a tests directory under the application root. Then create a Python file to store our tests (`test_flaskr.py`). When we format the filename like `test_*.py`, it will be auto-discoverable by pytest.

Next, we create a pytest fixture called `client()` that configures the application for testing and initializes a new database.:

```python
import os
import tempfile

import pytest

from flaskr import flaskr


@pytest.fixture
def client():
    db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
    flaskr.app.config['TESTING'] = True
    client = flaskr.app.test_client()

    with flaskr.app.app_context():
        flaskr.init_db()

    yield client

    os.close(db_fd)
    os.unlink(flaskr.app.config['DATABASE'])
```

This client fixture will be called by each individual test. It gives us a simple interface to the application, where we can trigger test requests to the application. The client will also keep track of cookies for us.

During setup, the `TESTING` config flag is activated. What this does is disable error catching during request handling, so that you get better error reports when performing test requests against the application.

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.

To delete the database after the test, the fixture closes the file and removes it from the filesystem.

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

```
$ pytest
```

```
=============== test session starts ===============
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 0 items


=========== no tests ran in 0.07 seconds ===========
```

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.

1.7.3 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 function to `test_flaskr.py`, like this:

```python
def test_empty_db(client):
    """Start with a blank database."""

    rv = client.get('/')
    assert b'No entries here so far' in rv.data
```

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

By using `client.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:

```
$ pytest -v

=============== test session starts ===============
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 1 items

tests/test_flaskr.py::test_empty_db PASSED

============= 1 passed in 0.10 seconds =============
```

1.7.4 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 functions to your `test_flaskr.py` file:

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


def logout(client):
    return client.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 function:

```python
def test_login_logout(client):
    """Make sure login and logout works."""

    rv = login(client, flaskr.app.config['USERNAME'], flaskr.app.config['PASSWORD
→'])
    assert b'You were logged in' in rv.data

    rv = logout(client)
    assert b'You were logged out' in rv.data

    rv = login(client, flaskr.app.config['USERNAME'] + 'x', flaskr.app.config[
→'PASSWORD'])
    assert b'Invalid username' in rv.data

    rv = login(client, flaskr.app.config['USERNAME'], flaskr.app.config['PASSWORD
→'] + 'x')
    assert b'Invalid password' in rv.data
```

1.7.5 Test Adding Messages

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

```python
def test_messages(client):
    """Test that messages work."""

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

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

Running that should now give us three passing tests:

```
$ pytest -v

================ test session starts ================
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 3 items

tests/test_flaskr.py::test_empty_db PASSED
tests/test_flaskr.py::test_login_logout PASSED
tests/test_flaskr.py::test_messages PASSED

============= 3 passed in 0.23 seconds ==============
```

1.7.6 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:

```python
import flask

app = flask.Flask(__name__)

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

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

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

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

```python
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.

1.7.7 Faking Resources and Context

New in version 0.10.

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

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

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

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

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

And then to use it:

```
from flask import json, jsonify
```

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


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

1.7.8 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).

1.7.9 Accessing and Modifying Sessions

New in version 0.8.

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

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

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

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

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

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

1.7.10 Testing JSON APIs

New in version 1.0.

Flask has great support for JSON, and is a popular choice for building JSON APIs. Making requests with JSON data and examining JSON data in responses is very convenient:

```
from flask import request, jsonify

@app.route('/api/auth')
def auth():
    json_data = request.get_json()
    email = json_data['email']
    password = json_data['password']
    return jsonify(token=generate_token(email, password))

with app.test_client() as c:
    rv = c.post('/api/auth', json={
        'username': 'flask', 'password': 'secret'
    })
    json_data = rv.get_json()
    assert verify_token(email, json_data['token'])
```

Passing the `json` argument in the test client methods sets the request data to the JSON-serialized object and sets the content type to `application/json`. You can get the JSON data from the request or response with `get_json`.

1.7.11 Testing CLI Commands

Click comes with utilities for testing your CLI commands. A `CliRunner` runs commands in isolation and captures the output in a `Result` object.

Flask provides `test_cli_runner()` to create a `FlaskCliRunner` that passes the Flask app to the CLI automatically. Use its `invoke()` method to call commands in the same way they would be called from the command line.

```
import click
```

```python
@app.cli.command('hello')
@click.option('--name', default='World')
def hello_command(name)
    click.echo(f'Hello, {name}!')


def test_hello():
    runner = app.test_cli_runner()

    # invoke the command directly
    result = runner.invoke(hello_command, ['--name', 'Flask'])
    assert 'Hello, Flask' in result.output

    # or by name
    result = runner.invoke(args=['hello'])
    assert 'World' in result.output
```

In the example above, invoking the command by name is useful because it verifies that the command was correctly registered with the app.

If you want to test how your command parses parameters, without running the command, use its `make_context()` method. This is useful for testing complex validation rules and custom types.

```python
def upper(ctx, param, value):
    if value is not None:
        return value.upper()


@app.cli.command('hello')
@click.option('--name', default='World', callback=upper)
def hello_command(name)
    click.echo(f'Hello, {name}!')


def test_hello_params():
    context = hello_command.make_context('hello', ['--name', 'flask'])
    assert context.params['name'] == 'FLASK'
```

1.8 Application Errors

New in version 0.3.

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

- the client terminated the request early and the application was still reading from the incoming data

- the database server was overloaded and could not handle the query

- a filesystem is full

- a harddrive crashed

- a backend server overloaded

- a programming error in a library you are using

- network connection of the server to another system failed

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

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

1.8.1 Error Logging Tools

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

To use Sentry you need to install the *raven* client with extra *flask* dependencies:

```
$ pip install raven[flask]
```

And then add this to your Flask app:

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

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

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

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

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

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

1.8.2 Error handlers

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

An error handler is a normal view function that return a response, but instead of being registered for a route, it is registered for an exception or HTTP status code that would is raised while trying to handle a request.

Registering

Register handlers by decorating a function with errorhandler(). Or use register_error_handler() to register the function later. Remember to set the error code when returning the response.

```python
@app.errorhandler(werkzeug.exceptions.BadRequest)
def handle_bad_request(e):
    return 'bad request!', 400

# or, without the decorator
app.register_error_handler(400, handle_bad_request)
```

werkzeug.exceptions.HTTPException subclasses like BadRequest and their HTTP codes are interchangeable when registering handlers. (BadRequest.code == 400)

Non-standard HTTP codes cannot be registered by code because they are not known by Werkzeug. Instead, define a subclass of HTTPException with the appropriate code and register and raise that exception class.

```python
class InsufficientStorage(werkzeug.exceptions.HTTPException):
    code = 507
    description = 'Not enough storage space.'

app.register_error_handler(InsuffcientStorage, handle_507)

raise InsufficientStorage()
```

Handlers can be registered for any exception class, not just HTTPException subclasses or HTTP status codes. Handlers can be registered for a specific class, or for all sub-classes of a parent class.

Handling

When an exception is caught by Flask while handling a request, it is first looked up by code. If no handler is registered for the code, it is looked up by its class hierarchy; the most specific handler is chosen. If no handler is registered, HTTPException subclasses show a generic message about their code, while other exceptions are converted to a generic 500 Internal Server Error.

For example, if an instance of `ConnectionRefusedError` is raised, and a handler is registered for `ConnectionError` and `ConnectionRefusedError`, the more specific `ConnectionRefusedError` handler is called with the exception instance to generate the response.

Handlers registered on the blueprint take precedence over those registered globally on the application, assuming a blueprint is handling the request that raises the exception. However, the blueprint cannot handle 404 routing errors because the 404 occurs at the routing level before the blueprint can be determined.

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

1.8.3 Logging

See *Logging* for information on how to log exceptions, such as by emailing them to admins.

1.9 Debugging Application Errors

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

1.9.1 When in Doubt, Run Manually

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

1.9.2 Working with Debuggers

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

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

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

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

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

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

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

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

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

1.10 Logging

Flask uses standard Python `logging`. All Flask-related messages are logged under the `'flask'` logger namespace. `Flask.logger` returns the logger named `'flask.app'`, and can be used to log messages for your application.

```python
@app.route('/login', methods=['POST'])
def login():
    user = get_user(request.form['username'])

    if user.check_password(request.form['password']):
        login_user(user)
        app.logger.info('%s logged in successfully', user.username)
        return redirect(url_for('index'))
    else:
        app.logger.info('%s failed to log in', user.username)
        abort(401)
```

1.10.1 Basic Configuration

When you want to configure logging for your project, you should do it as soon as possible when the program starts. If `app.logger` is accessed before logging is configured, it will add a default handler. If possible, configure logging before creating the application object.

This example uses `dictConfig()` to create a logging configuration similar to Flask's default, except for all logs:

```python
from logging.config import dictConfig

dictConfig({
    'version': 1,
    'formatters': {'default': {
        'format': '[%(asctime)s] %(levelname)s in %(module)s: %(message)s',
    }},
    'handlers': {'wsgi': {
        'class': 'logging.StreamHandler',
        'stream': 'ext://flask.logging.wsgi_errors_stream',
        'formatter': 'default'
    }},
    'root': {
        'level': 'INFO',
        'handlers': ['wsgi']
    }
})

app = Flask(__name__)
```

Default Configuration

If you do not configure logging yourself, Flask will add a `StreamHandler` to `app.logger` automatically. During requests, it will write to the stream specified by the WSGI server in `environ['wsgi.errors']` (which is usually `sys.stderr`). Outside a request, it will log to `sys.stderr`.

Removing the Default Handler

If you configured logging after accessing `app.logger`, and need to remove the default handler, you can import and remove it:

```python
from flask.logging import default_handler

app.logger.removeHandler(default_handler)
```

1.10.2 Email Errors to Admins

When running the application on a remote server for production, you probably won't be looking at the log messages very often. The WSGI server will probably send log messages to a file, and you'll only check that file if a user tells you something went wrong.

To be proactive about discovering and fixing bugs, you can configure a `logging.handlers.SMTPHandler` to send an email when errors and higher are logged.

```python
import logging
from logging.handlers import SMTPHandler

mail_handler = SMTPHandler(
    mailhost='127.0.0.1',
    fromaddr='server-error@example.com',
    toaddrs=['admin@example.com'],
    subject='Application Error'
)
mail_handler.setLevel(logging.ERROR)
mail_handler.setFormatter(logging.Formatter(
    '[%(asctime)s] %(levelname)s in %(module)s: %(message)s'
))

if not app.debug:
    app.logger.addHandler(mail_handler)
```

This requires that you have an SMTP server set up on the same server. See the Python docs for more information about configuring the handler.

1.10.3 Injecting Request Information

Seeing more information about the request, such as the IP address, may help debugging some errors. You can subclass `logging.Formatter` to inject your own fields that can be used in messages. You can change the formatter for Flask's default handler, the mail handler defined above, or any other handler.

```python
from flask import request
from flask.logging import default_handler

class RequestFormatter(logging.Formatter):
    def format(self, record):
        record.url = request.url
        record.remote_addr = request.remote_addr
        return super().format(record)

formatter = RequestFormatter(
    '[%(asctime)s] %(remote_addr)s requested %(url)s\n'
    '%(levelname)s in %(module)s: %(message)s'
)
```

```
default_handler.setFormatter(formatter)
mail_handler.setFormatter(formatter)
```

1.10.4 Other Libraries

Other libraries may use logging extensively, and you want to see relevant messages from those logs too. The simplest way to do this is to add handlers to the root logger instead of only the app logger.

```python
from flask.logging import default_handler

root = logging.getLogger()
root.addHandler(default_handler)
root.addHandler(mail_handler)
```

Depending on your project, it may be more useful to configure each logger you care about separately, instead of configuring only the root logger.

```python
for logger in (
    app.logger,
    logging.getLogger('sqlalchemy'),
    logging.getLogger('other_package'),
):
    logger.addHandler(default_handler)
    logger.addHandler(mail_handler)
```

Werkzeug

Werkzeug logs basic request/response information to the 'werkzeug' logger. If the root logger has no handlers configured, Werkzeug adds a `StreamHandler` to its logger.

Flask Extensions

Depending on the situation, an extension may choose to log to `app.logger` or its own named logger. Consult each extension's documentation for details.

1.11 Configuration Handling

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

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

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

1.11.1 Configuration Basics

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

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

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

```python
app.testing = True
```

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

```python
app.config.update(
    TESTING=True,
    SECRET_KEY=b'_5#y2L"F4Q8z\n\xec]/'
)
```

1.11.2 Environment and Debug Features

The `ENV` and `DEBUG` config values are special because they may behave inconsistently if changed after the app has begun setting up. In order to set the environment and debug mode reliably, Flask uses environment variables.

The environment is used to indicate to Flask, extensions, and other programs, like Sentry, what context Flask is running in. It is controlled with the `FLASK_ENV` environment variable and defaults to `production`.

Setting `FLASK_ENV` to `development` will enable debug mode. `flask run` will use the interactive debugger and reloader by default in debug mode. To control this separately from the environment, use the `FLASK_DEBUG` flag.

Changed in version 1.0: Added `FLASK_ENV` to control the environment separately from debug mode. The development environment enables debug mode.

To switch Flask to the development environment and enable debug mode, set `FLASK_ENV`:

```
$ export FLASK_ENV=development
$ flask run
```

(On Windows, use `set` instead of `export`.)

Using the environment variables as described above is recommended. While it is possible to set `ENV` and `DEBUG` in your config or code, this is strongly discouraged. They can't be read early by the `flask` command, and some systems or extensions may have already configured themselves based on a previous value.

1.11.3 Builtin Configuration Values

The following configuration values are used internally by Flask:

ENV

> What environment the app is running in. Flask and extensions may enable behaviors based on the environment, such as enabling debug mode. The env attribute maps to this config key. This is set by the `FLASK_ENV` environment variable and may not behave as expected if set in code.
>
> **Do not enable development when deploying in production.**
>
> Default: `'production'`
>
> New in version 1.0.

DEBUG

> Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. The debug attribute maps to this config key. This is enabled when `ENV` is `'development'` and is overridden by the `FLASK_DEBUG` environment variable. It may not behave as expected if set in code.
>
> **Do not enable debug mode when deploying in production.**
>
> Default: `True` if `ENV` is `'production'`, or `False` otherwise.

TESTING

> Enable testing mode. Exceptions are propagated rather than handled by the the app's error handlers. Extensions may also change their behavior to facilitate easier testing. You should enable this in your own tests.
>
> Default: `False`

PROPAGATE_EXCEPTIONS

> Exceptions are re-raised rather than being handled by the app's error handlers. If not set, this is implicitly true if `TESTING` or `DEBUG` is enabled.
>
> Default: `None`

PRESERVE_CONTEXT_ON_EXCEPTION

> Don't pop the request context when an exception occurs. If not set, this is true if `DEBUG` is true. This allows debuggers to introspect the request data on errors, and should normally not need to be set directly.
>
> Default: `None`

TRAP_HTTP_EXCEPTIONS

If there is no handler for an `HTTPException`-type exception, re-raise it to be handled by the interactive debugger instead of returning it as a simple error response.

Default: `False`

TRAP_BAD_REQUEST_ERRORS

Trying to access a key that doesn't exist from request dicts like `args` and `form` will return a 400 Bad Request error page. Enable this to treat the error as an unhandled exception instead so that you get the interactive debugger. This is a more specific version of `TRAP_HTTP_EXCEPTIONS`. If unset, it is enabled in debug mode.

Default: `None`

SECRET_KEY

A secret key that will be used for securely signing the session cookie and can be used for any other security related needs by extensions or your application. It should be a long random string of bytes, although unicode is accepted too. For example, copy the output of this to your config:

```
python -c 'import os; print(os.urandom(16))'
b'_5#y2L"F4Q8z\n\xec]/'
```

Do not reveal the secret key when posting questions or committing code.

Default: `None`

SESSION_COOKIE_NAME

The name of the session cookie. Can be changed in case you already have a cookie with the same name.

Default: `'session'`

SESSION_COOKIE_DOMAIN

The domain match rule that the session cookie will be valid for. If not set, the cookie will be valid for all subdomains of `SERVER_NAME`. If `False`, the cookie's domain will not be set.

Default: `None`

SESSION_COOKIE_PATH

The path that the session cookie will be valid for. If not set, the cookie will be valid underneath `APPLICATION_ROOT` or / if that is not set.

Default: `None`

SESSION_COOKIE_HTTPONLY

Browsers will not allow JavaScript access to cookies marked as "HTTP only" for security.

Default: `True`

SESSION_COOKIE_SECURE

> Browsers will only send cookies with requests over HTTPS if the cookie is marked "secure". The application must be served over HTTPS for this to make sense.
>
> Default: `False`

SESSION_COOKIE_SAMESITE

> Restrict how cookies are sent with requests from external sites. Can be set to `'Lax'` (recommended) or `'Strict'`. See *Set-Cookie options*.
>
> Default: `None`
>
> New in version 1.0.

PERMANENT_SESSION_LIFETIME

> If `session.permanent` is true, the cookie's expiration will be set this number of seconds in the future. Can either be a `datetime.timedelta` or an `int`.
>
> Flask's default cookie implementation validates that the cryptographic signature is not older than this value.
>
> Default: `timedelta(days=31)` (2678400 seconds)

SESSION_REFRESH_EACH_REQUEST

> Control whether the cookie is sent with every response when `session.permanent` is true. Sending the cookie every time (the default) can more reliably keep the session from expiring, but uses more bandwidth. Non-permanent sessions are not affected.
>
> Default: `True`

USE_X_SENDFILE

> When serving files, set the `X-Sendfile` header instead of serving the data with Flask. Some web servers, such as Apache, recognize this and serve the data more efficiently. This only makes sense when using such a server.
>
> Default: `False`

SEND_FILE_MAX_AGE_DEFAULT

> When serving files, set the cache control max age to this number of seconds. Can either be a `datetime.timedelta` or an `int`. Override this value on a per-file basis using `get_send_file_max_age()` on the application or blueprint.
>
> Default: `timedelta(hours=12)` (43200 seconds)

SERVER_NAME

> Inform the application what host and port it is bound to. Required for subdomain route matching support.
>
> If set, will be used for the session cookie domain if `SESSION_COOKIE_DOMAIN` is not set. Modern web browsers will not allow setting cookies for domains without a dot. To use a domain locally, add any names that should route to the app to your `hosts` file.

```
127.0.0.1 localhost.dev
```

If set, `url_for` can generate external URLs with only an application context instead of a request context.

Default: `None`

APPLICATION_ROOT
Inform the application what path it is mounted under by the application / web server.

Will be used for the session cookie path if `SESSION_COOKIE_PATH` is not set.

Default: `'/'`

PREFERRED_URL_SCHEME
Use this scheme for generating external URLs when not in a request context.

Default: `'http'`

MAX_CONTENT_LENGTH
Don't read more than this many bytes from the incoming request data. If not set and the request does not specify a `CONTENT_LENGTH`, no data will be read for security.

Default: `None`

JSON_AS_ASCII
Serialize objects to ASCII-encoded JSON. If this is disabled, the JSON will be returned as a Unicode string, or encoded as `UTF-8` by `jsonify`. This has security implications when rendering the JSON in to JavaScript in templates, and should typically remain enabled.

Default: `True`

JSON_SORT_KEYS
Sort the keys of JSON objects alphabetically. This is useful for caching because it ensures the data is serialized the same way no matter what Python's hash seed is. While not recommended, you can disable this for a possible performance improvement at the cost of caching.

Default: `True`

JSONIFY_PRETTYPRINT_REGULAR
`jsonify` responses will be output with newlines, spaces, and indentation for easier reading by humans. Always enabled in debug mode.

Default: `False`

JSONIFY_MIMETYPE
The mimetype of `jsonify` responses.

Default: `'application/json'`

TEMPLATES_AUTO_RELOAD

> Reload templates when they are changed. If not set, it will be enabled in debug mode.

> Default: `None`

EXPLAIN_TEMPLATE_LOADING

> Log debugging information tracing how a template file was loaded. This can be useful to figure out why a template was not loaded or the wrong file appears to be loaded.

> Default: `False`

MAX_COOKIE_SIZE

> Warn if cookie headers are larger than this many bytes. Defaults to `4093`. Larger cookies may be silently ignored by browsers. Set to `0` to disable the warning.

New in version 0.4: `LOGGER_NAME`

New in version 0.5: `SERVER_NAME`

New in version 0.6: `MAX_CONTENT_LENGTH`

New in version 0.7: `PROPAGATE_EXCEPTIONS`, `PRESERVE_CONTEXT_ON_EXCEPTION`

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

New in version 0.9: `PREFERRED_URL_SCHEME`

New in version 0.10: `JSON_AS_ASCII`, `JSON_SORT_KEYS`, `JSONIFY_PRETTYPRINT_REGULAR`

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

Changed in version 1.0: `LOGGER_NAME` and `LOGGER_HANDLER_POLICY` were removed. See *Logging* for information about configuration.

Added `ENV` to reflect the `FLASK_ENV` environment variable.

Added `SESSION_COOKIE_SAMESITE` to control the session cookie's `SameSite` option.

Added `MAX_COOKIE_SIZE` to control a warning from Werkzeug.

1.11.4 Configuring from Files

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

So a common pattern is this:

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

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

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

On Windows systems use the *set* builtin instead:

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

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

Here is an example of a configuration file:

```
# Example configuration
DEBUG = False
SECRET_KEY = b'_5#y2L"F4Q8z\n\xec]/'
```

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.

1.11.5 Configuring from Environment Variables

In addition to pointing to configuration files using environment variables, you may find it useful (or necessary) to control your configuration values directly from the environment.

Environment variables can be set on Linux or OS X with the export command in the shell before starting the server:

```
$ export SECRET_KEY='5f352379324c22463451387a0aec5d2f'
$ export DEBUG=False
$ python run-app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader...
```

On Windows systems use the *set* builtin instead:

```
>set SECRET_KEY='5f352379324c22463451387a0aec5d2f'
>set DEBUG=False
```

While this approach is straightforward to use, it is important to remember that environment variables are strings – they are not automatically deserialized into Python types.

Here is an example of a configuration file that uses environment variables:

```
# Example configuration
import os

ENVIRONMENT_DEBUG = os.environ.get("DEBUG", default=False)
if ENVIRONMENT_DEBUG.lower() in ("f", "false"):
    ENVIRONMENT_DEBUG = False

DEBUG = ENVIRONMENT_DEBUG
SECRET_KEY = os.environ.get("SECRET_KEY", default=None)
if not SECRET_KEY:
    raise ValueError("No secret key set for Flask application")
```

Notice that any value besides an empty string will be interpreted as a boolean `True` value in Python, which requires care if an environment explicitly sets values intended to be `False`.

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` class documentation.

1.11.6 Configuration Best Practices

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

1. Create your application in a function and register blueprints on it. That way you can create multiple instances of your application with different configurations attached which makes unittesting a lot easier. You can use this to pass in configuration as needed.

2. Do not write code that needs the configuration at import time. If you limit yourself to request-only accesses to the configuration you can reconfigure the object later on as needed.

1.11.7 Development / Production

Most applications need more than one configuration. There should be at least separate configurations for the production server and the one used during development. The

easiest way to handle this is to use a default configuration that is always loaded and part of the version control, and a separate configuration that overrides the values as necessary as mentioned in the example above:

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

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

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

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

```python
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()`:

```python
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.

1.11.8 Instance Folders

New in version 0.8.

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

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

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

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

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

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

- Uninstalled module:

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

- Uninstalled package:

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

- Installed module or package:

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

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

Since the config object provided loading of configuration files from relative filenames we made it possible to change the loading via filenames to be relative to the instance path if wanted. The behavior of relative paths in config files can be flipped between

"relative to the application root" (the default) to "relative to instance folder" via the *instance_relative_config* switch to the application constructor:

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

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

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

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

Example usage for both:

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

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

1.12 Signals

New in version 0.6.

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

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

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

The big advantage of signals over handlers is that you can safely subscribe to them for just a split second. These temporary subscriptions are helpful for unit testing for

example. Say you want to know what templates were rendered as part of a request: signals allow you to do exactly that.

1.12.1 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 from all applications. This is especially true if you are developing an extension.

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

```python
from flask import template_rendered
from contextlib import contextmanager

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

This can now easily be paired with a test client:

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

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

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

Additionally there is a convenient helper method (`connected_to()`) that allows you to temporarily subscribe a function to a signal with a context manager on its own.

Because the return value of the context manager cannot be specified that way, you have to pass the list in as an argument:

```
from flask import template_rendered

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

The example above would then look like this:

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

Blinker API Changes

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

1.12.2 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 want to gracefully degrade for missing blinker installations, you can do so by using the `flask.signals.Namespace` class.

1.12.3 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:

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

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

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

Passing Proxies as Senders

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

1.12.4 Signals and Flask's Request Context

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

1.12.5 Decorator Based Signal Subscriptions

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

```python
from flask import template_rendered

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

1.12.6 Core Signals

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

1.13 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.

1.13.1 Basic Principle

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

```python
@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:

```python
from flask.views import View

class ShowUsers(View):

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

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

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

```python
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'))
```

1.13.2 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'))
```

1.13.3 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):

```python
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.

1.13.4 Decorating Views

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

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

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

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

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

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

1.13.5 Method Views for APIs

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

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

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

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

```python
class UserAPI(MethodView):

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

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

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

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

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

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

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

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

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

1.14 The Application Context

The application context keeps track of the application-level data during a request, CLI command, or other activity. Rather than passing the application around to each function, the `current_app` and `g` proxies are accessed instead.

This is similar to the *The Request Context*, which keeps track of request-level data during a request. A corresponding application context is pushed when a request context is pushed.

1.14.1 Purpose of the Context

The `Flask` application object has attributes, such as `config`, that are useful to access within views and *CLI commands*. However, importing the app instance within the modules in your project is prone to circular import issues. When using the *app factory pattern* or writing reusable *blueprints* or *extensions* there won't be an app instance to import at all.

Flask solves this issue with the *application context*. Rather than referring to an app directly, you use the the `current_app` proxy, which points to the application handling the current activity.

Flask automatically *pushes* an application context when handling a request. View functions, error handlers, and other functions that run during a request will have access to `current_app`.

Flask will also automatically push an app context when running CLI commands registered with `Flask.cli` using `@app.cli.command()`.

1.14.2 Lifetime of the Context

The application context is created and destroyed as necessary. When a Flask application begins handling a request, it pushes an application context and a *request context*. When the request ends it pops the request context then the application context. Typically, an application context will have the same lifetime as a request.

See *The Request Context* for more information about how the contexts work and the full lifecycle of a request.

1.14.3 Manually Push a Context

If you try to access `current_app`, or anything that uses it, outside an application context, you'll get this error message:

```
RuntimeError: Working outside of application context.

This typically means that you attempted to use functionality that
needed to interface with the current application object in some way.
To solve this, set up an application context with app.app_context().
```

If you see that error while configuring your application, such as when initializing an extension, you can push a context manually since you have direct access to the app. Use `app_context()` in a `with` block, and everything that runs in the block will have access to `current_app`.

```python
def create_app():
    app = Flask(__name__)

    with app.app_context():
        init_db()

    return app
```

If you see that error somewhere else in your code not related to configuring the application, it most likely indicates that you should move that code into a view function or CLI command.

1.14.4 Storing Data

The application context is a good place to store common data during a request or CLI command. Flask provides the `g` object for this purpose. It is a simple namespace object that has the same lifetime as an application context.

Note: The g name stands for "global", but that is referring to the data being global *within a context*. The data on g is lost after the context ends, and it is not an appropriate place to store data between requests. Use the `session` or a database to store data across requests.

A common use for `g` is to manage resources during a request.

1. `get_X()` creates resource X if it does not exist, caching it as `g.X`.

2. `teardown_X()` closes or otherwise deallocates the resource if it exists. It is registered as a `teardown_appcontext()` handler.

For example, you can manage a database connection using this pattern:

```python
from flask import g

def get_db():
    if 'db' not in g:
        g.db = connect_to_database()

    return g.db

@app.teardown_appcontext
def teardown_db():
    db = g.pop('db', None)

    if db is not None:
        db.close()
```

During a request, every call to `get_db()` will return the same connection, and it will be closed automatically at the end of the request.

You can use `LocalProxy` to make a new context local from `get_db()`:

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

Accessing `db` will call `get_db` internally, in the same way that `current_app` works.

If you're writing an extension, g should be reserved for user code. You may store internal data on the context itself, but be sure to use a sufficiently unique name. The current context is accessed with `_app_ctx_stack.top`. For more information see *Flask Extension Development*.

1.14.5 Events and Signals

The application will call functions registered with `teardown_appcontext()` when the application context is popped.

If `signals_available` is true, the following signals are sent: `appcontext_pushed`, `appcontext_tearing_down`, and `appcontext_popped`.

1.15 The Request Context

The request context keeps track of the request-level data during a request. Rather than passing the request object to each function that runs during a request, the `request` and `session` proxies are accessed instead.

This is similar to the *The Application Context*, which keeps track of the application-level data independent of a request. A corresponding application context is pushed when a request context is pushed.

1.15.1 Purpose of the Context

When the Flask application handles a request, it creates a Request object based on the environment it received from the WSGI server. Because a *worker* (thread, process, or coroutine depending on the server) handles only one request at a time, the request data can be considered global to that worker during that request. Flask uses the term *context local* for this.

Flask automatically *pushes* a request context when handling a request. View functions, error handlers, and other functions that run during a request will have access to the request proxy, which points to the request object for the current request.

1.15.2 Lifetime of the Context

When a Flask application begins handling a request, it pushes a request context, which also pushes an *The Application Context*. When the request ends it pops the request context then the application context.

The context is unique to each thread (or other worker type). request cannot be passed to another thread, the other thread will have a different context stack and will not know about the request the parent thread was pointing to.

Context locals are implemented in Werkzeug. See Context Locals for more information on how this works internally.

1.15.3 Manually Push a Context

If you try to access request, or anything that uses it, outside a request context, you'll get this error message:

```
RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that
needed an active HTTP request. Consult the documentation on testing
for information about how to avoid this problem.
```

This should typically only happen when testing code that expects an active request. One option is to use the test client to simulate a full request. Or you can use test_request_context() in a with block, and everything that runs in the block will have access to request, populated with your test data.

```
def generate_report(year):
    format = request.args.get('format')
    ...
```

```
with app.test_request_context(
        '/make_report/2017', data={'format': 'short'}):
    generate_report()
```

If you see that error somewhere else in your code not related to testing, it most likely indicates that you should move that code into a view function.

For information on how to use the request context from the interactive Python shell, see *Working with the Shell*.

1.15.4 How the Context Works

The `Flask.wsgi_app()` method is called to handle each request. It manages the contexts during the request. Internally, the request and application contexts work as stacks, `_request_ctx_stack` and `_app_ctx_stack`. When contexts are pushed onto the stack, the proxies that depend on them are available and point at information from the top context on the stack.

When the request starts, a `RequestContext` is created and pushed, which creates and pushes an `AppContext` first if a context for that application is not already the top context. While these contexts are pushed, the `current_app`, `g`, `request`, and `session` proxies are available to the original thread handling the request.

Because the contexts are stacks, other contexts may be pushed to change the proxies during a request. While this is not a common pattern, it can be used in advanced applications to, for example, do internal redirects or chain different applications together.

After the request is dispatched and a response is generated and sent, the request context is popped, which then pops the application context. Immediately before they are popped, the `teardown_request()` and `teardown_appcontext()` functions are are executed. These execute even if an unhandled exception occurred during dispatch.

1.15.5 Callbacks and Errors

Flask dispatches a request in multiple stages which can affect the request, response, and how errors are handled. The contexts are active during all of these stages.

A `Blueprint` can add handlers for these events that are specific to the blueprint. The handlers for a blueprint will run if the blueprint owns the route that matches the request.

1. Before each request, `before_request()` functions are called. If one of these functions return a value, the other functions are skipped. The return value is treated as the response and the view function is not called.

2. If the `before_request()` functions did not return a response, the view function for the matched route is called and returns a response.

3. The return value of the view is converted into an actual response object and passed to the `after_request()` functions. Each function returns a modified or new response object.

4. After the response is returned, the contexts are popped, which calls the `teardown_request()` and `teardown_appcontext()` functions. These functions are called even if an unhandled exception was raised at any point above.

If an exception is raised before the teardown functions, Flask tries to match it with an `errorhandler()` function to handle the exception and return a response. If no error handler is found, or the handler itself raises an exception, Flask returns a generic `500 Internal Server Error` response. The teardown functions are still called, and are passed the exception object.

If debug mode is enabled, unhandled exceptions are not converted to a `500` response and instead are propagated to the WSGI server. This allows the development server to present the interactive debugger with the traceback.

Teardown Callbacks

The teardown callbacks are independent of the request dispatch, and are instead called by the contexts when they are popped. The functions are called even if there is an unhandled exception during dispatch, and for manually pushed contexts. This means there is no guarantee that any other parts of the request dispatch have run first. Be sure to write these functions in a way that does not depend on other callbacks and will not fail.

During testing, it can be useful to defer popping the contexts after the request ends, so that their data can be accessed in the test function. Using the `test_client()` as a `with` block to preserve the contexts until the with block exits.

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def hello():
    print('during view')
    return 'Hello, World!'

@app.teardown_request
def show_teardown(exception):
    print('after with block')

with app.test_request_context():
    print('during with block')

# teardown functions are called after the context with block exits

with app.test_client():
    client.get('/')
```

```
    # the contexts are not popped even though the request ended
    print(request.path)

# the contexts are popped and teardown functions are called after
# the client with block exists
```

Signals

If `signals_available` is true, the following signals are sent:

1. `request_started` is sent before the `before_request()` functions are called.

2. `request_finished` is sent after the `after_request()` functions are called.

3. `got_request_exception` is sent when an exception begins to be handled, but before an `errorhandler()` is looked up or called.

4. `request_tearing_down` is sent after the `teardown_request()` functions are called.

1.15.6 Context Preservation on Error

At the end of a request, the request context is popped and all data associated with it is destroyed. If an error occurs during development, it is useful to delay destroying the data for debugging purposes.

When the development server is running in development mode (the `FLASK_ENV` environment variable is set to `'development'`), the error and data will be preserved and shown in the interactive debugger.

This behavior can be controlled with the `PRESERVE_CONTEXT_ON_EXCEPTION` config. As described above, it defaults to `True` in the development environment.

Do not enable `PRESERVE_CONTEXT_ON_EXCEPTION` in production, as it will cause your application to leak memory on exceptions.

1.15.7 Notes On Proxies

Some of the objects provided by Flask are proxies to other objects. The proxies are accessed in the same way for each worker thread, but point to the unique object bound to each worker behind the scenes as described on this page.

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 cannot fake their type as the actual object types. If you want to perform instance checks, you have to do that on the object being proxied.

• If the specific object reference is important, for example for sending *Signals* or passing data to a background thread.

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

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

1.16 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.

1.16.1 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.

1.16.2 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.

1.16.3 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:

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

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

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

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

1.16.4 Registering Blueprints

So how do you register that blueprint? Like this:

```python
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.

1.16.5 Blueprint Resources

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

Blueprint Resource Folder

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

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

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

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

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

Static Files

A blueprint can expose a folder with static files by providing the path to the folder on the filesystem with the `static_folder` argument. It is either an absolute path or

relative to the blueprint's location:

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

By default the rightmost part of the path is where it is exposed on the web. This can be changed with the `static_url` argument. Because the folder is called `static` here it will be available at the `url_prefix` of the blueprint + `/static`. If the blueprint has the prefix `/admin`, the static URL will be `/admin/static`.

The endpoint is named `blueprint_name.static`. You can generate URLs to it with `url_for()` like you would with the static folder of the application:

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

However, if the blueprint does not have a `url_prefix`, it is not possible to access the blueprint's static folder. This is because the URL would be `/static` in this case, and the application's `/static` route takes precedence. Unlike template folders, blueprint static folders are not searched if the file does not exist in the application static folder.

Templates

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

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

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

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

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

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

```
yourpackage/
    blueprints/
        admin/
            templates/
```

```
            admin/
                index.html
            __init__.py
```

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

1.16.6 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.

1.16.7 Error Handlers

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

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

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

Most errorhandlers will simply work as expected; however, there is a caveat concerning handlers for 404 and 405 exceptions. These errorhandlers are only invoked from an appropriate `raise` statement or a call to `abort` in another of the blueprint's view functions; they are not invoked by, e.g., an invalid URL access. This is because the blueprint does not "own" a certain URL space, so the application instance has no way of knowing which blueprint errorhandler it should run if given an invalid URL. If you would like to execute different handling strategies for these errors based on URL prefixes, they may be defined at the application level using the `request` proxy object:

```python
@app.errorhandler(404)
@app.errorhandler(405)
def _handle_api_error(ex):
    if request.path.startswith('/api/'):
        return jsonify_error(ex)
    else:
        return ex
```

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

1.17 Extensions

Extensions are extra packages that add functionality to a Flask application. For example, an extension might add support for sending email or connecting to a database. Some extensions add entire new frameworks to help build certain types of applications, like a ReST API.

1.17.1 Finding Extensions

Flask extensions are usually named "Flask-Foo" or "Foo-Flask". Many extensions are listed in the Extension Registry, which can be updated by extension developers. You can also search PyPI for packages tagged with Framework :: Flask.

1.17.2 Using Extensions

Consult each extension's documentation for installation, configuration, and usage instructions. Generally, extensions pull their own configuration from `app.config` and are passed an application instance during initialization. For example, an extension caled "Flask-Foo" might be used like this:

```python
from flask_foo import Foo

foo = Foo()

app = Flask(__name__)
app.config.update(
    FOO_BAR='baz',
    FOO_SPAM='eggs',
)

foo.init_app(app)
```

1.17.3 Building Extensions

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

1.18 Command Line Interface

Installing Flask installs the `flask` script, a Click command line interface, in your virtualenv. Executed from the terminal, this script gives access to built-in, extension, and application-defined commands. The `--help` option will give more information about any commands and options.

1.18.1 Application Discovery

The `flask` command is installed by Flask, not your application; it must be told where to find your application in order to use it. The FLASK_APP environment variable is used to specify how to load the application.

Unix Bash (Linux, Mac, etc.):

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

Windows CMD:

```
> set FLASK_APP=hello
> flask run
```

Windows PowerShell:

```
> $env:FLASK_APP = "hello"
> flask run
```

While FLASK_APP supports a variety of options for specifying your application, most use cases should be simple. Here are the typical values:

(nothing) The file `wsgi.py` is imported, automatically detecting an app (app). This provides an easy way to create an app from a factory with extra arguments.

FLASK_APP=hello The name is imported, automatically detecting an app (app) or factory (`create_app`).

FLASK_APP has three parts: an optional path that sets the current working directory, a Python file or dotted import path, and an optional variable name of the instance or factory. If the name is a factory, it can optionally be followed by arguments in parentheses. The following values demonstrate these parts:

FLASK_APP=src/hello Sets the current working directory to `src` then imports `hello`.

FLASK_APP=hello.web Imports the path `hello.web`.

FLASK_APP=hello:app2 Uses the app2 Flask instance in `hello`.

FLASK_APP="hello:create_app('dev')" The `create_app` factory in `hello` is called with the string `'dev'` as the argument.

If `FLASK_APP` is not set, the command will look for a file called `wsgi.py` or `app.py` and try to detect an application instance or factory.

Within the given import, the command looks for an application instance named app or `application`, then any application instance. If no instance is found, the command looks for a factory function named `create_app` or `make_app` that returns an instance.

When calling an application factory, if the factory takes an argument named `info`, then the `ScriptInfo` instance is passed as a keyword argument. If parentheses follow the factory name, their contents are parsed as Python literals and passes as arguments to the function. This means that strings must still be in quotes.

1.18.2 Run the Development Server

The `run` command will start the development server. It replaces the `Flask.run()` method in most cases.

```
$ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

> **Warning:** Do not use this command to run your application in production. Only use the development server during development. The development server is provided for convenience, but is not designed to be particularly secure, stable, or efficient. See *Deployment Options* for how to run in production.

1.18.3 Open a Shell

To explore the data in your application, you can start an interactive Python shell with the `shell` command. An application context will be active, and the app instance will be imported.

```
$ flask shell
Python 3.6.2 (default, Jul 20 2017, 03:52:27)
[GCC 7.1.1 20170630] on linux
App: example
Instance: /home/user/Projects/hello/instance
>>>
```

Use `shell_context_processor()` to add other automatic imports.

1.18.4 Environments

New in version 1.0.

The environment in which the Flask app runs is set by the `FLASK_ENV` environment variable. If not set it defaults to `production`. The other recognized environment is `development`. Flask and extensions may choose to enable behaviors based on the environment.

If the env is set to `development`, the `flask` command will enable debug mode and `flask run` will enable the interactive debugger and reloader.

```
$ FLASK_ENV=development flask run
 * Serving Flask app "hello"
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with inotify reloader
 * Debugger is active!
 * Debugger PIN: 223-456-919
```

1.18.5 Debug Mode

Debug mode will be enabled when `FLASK_ENV` is `development`, as described above. If you want to control debug mode separately, use `FLASK_DEBUG`. The value `1` enables it, `0` disables it.

1.18.6 Environment Variables From dotenv

Rather than setting `FLASK_APP` each time you open a new terminal, you can use Flask's dotenv support to set environment variables automatically.

If python-dotenv is installed, running the `flask` command will set environment variables defined in the files `.env` and `.flaskenv`. This can be used to avoid having to set `FLASK_APP` manually every time you open a new terminal, and to set configuration using environment variables similar to how some deployment services work.

Variables set on the command line are used over those set in `.env`, which are used over those set in `.flaskenv`. `.flaskenv` should be used for public variables, such as `FLASK_APP`, while `.env` should not be committed to your repository so that it can set private variables.

Directories are scanned upwards from the directory you call `flask` from to locate the files. The current working directory will be set to the location of the file, with the assumption that that is the top level project directory.

The files are only loaded by the `flask` command or calling `run()`. If you would like to load these files when running in production, you should call `load_dotenv()` manually.

1.18.7 Environment Variables From virtualenv

If you do not want to install dotenv support, you can still set environment variables by adding them to the end of the virtualenv's `activate` script. Activating the virtualenv will set the variables.

Unix Bash, `venv/bin/activate`:

```
export FLASK_APP=hello
```

Windows CMD, `venv\Scripts\activate.bat`:

```
set FLASK_APP=hello
```

It is preferred to use dotenv support over this, since `.flaskenv` can be committed to the repository so that it works automatically wherever the project is checked out.

1.18.8 Custom Commands

The `flask` command is implemented using Click. See that project's documentation for full information about writing commands.

This example adds the command `create_user` that takes the argument `name`.

```python
import click
from flask import Flask

app = Flask(__name__)

@app.cli.command()
@click.argument('name')
def create_user(name):
    ...
```

```
flask create_user admin
```

This example adds the same command, but as `user create`, a command in a group. This is useful if you want to organize multiple related commands.

```python
import click
from flask import Flask
from flask.cli import AppGroup

app = Flask(__name__)
user_cli = AppGroup('user')
```

```
@user_cli.command('create')
@click.argument('name')
def create_user(name):
    ...


app.cli.add_command(user_cli)
```

```
flask user create demo
```

See *Testing CLI Commands* for an overview of how to test your custom commands.

Application Context

Commands added using the Flask app's cli command() decorator will be executed with an application context pushed, so your command and extensions have access to the app and its configuration. If you create a command using the Click command() decorator instead of the Flask decorator, you can use with_appcontext() to get the same behavior.

```
import click
from flask.cli import with_appcontext

@click.command
@with_appcontext
def do_work():
    ...


app.cli.add_command(do_work)
```

If you're sure a command doesn't need the context, you can disable it:

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

1.18.9 Plugins

Flask will automatically load commands specified in the flask.commands entry point. This is useful for extensions that want to add commands when they are installed. Entry points are specified in setup.py

```
from setuptools import setup

setup(
    name='flask-my-extension',
    ...,
    entry_points={
```

```
        'flask.commands': [
            'my-command=flask_my_extension.commands:cli'
        ],
    },
)
```

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

```python
import click

@click.command()
def cli():
    ...
```

Once that package is installed in the same virtualenv as your Flask project, you can run `flask my-command` to invoke the command.

1.18.10 Custom Scripts

When you are using the app factory pattern, it may be more convenient to define your own Click script. Instead of using `FLASK_APP` and letting Flask load your application, you can create your own Click object and export it as a console script entry point.

Create an instance of `FlaskGroup` and pass it the factory:

```python
import click
from flask import Flask
from flask.cli import FlaskGroup

def create_app():
    app = Flask('wiki')
    # other setup
    return app

@click.group(cls=FlaskGroup, create_app=create_app)
def cli():
    """Management script for the Wiki application."""
```

Define the entry point in `setup.py`:

```python
from setuptools import setup

setup(
    name='flask-my-extension',
    ...,
    entry_points={
        'console_scripts': [
            'wiki=wiki:cli'
        ],
```

```
    },
)
```

Install the application in the virtualenv in editable mode and the custom script is available. Note that you don't need to set `FLASK_APP`.

```
$ pip install -e .
$ wiki run
```

Errors in Custom Scripts

When using a custom script, if you introduce an error in your module-level code, the reloader will fail because it can no longer load the entry point.

The `flask` command, being separate from your code, does not have this issue and is recommended in most cases.

1.18.11 PyCharm Integration

Prior to PyCharm 2018.1, the Flask CLI features weren't yet fully integrated into PyCharm. We have to do a few tweaks to get them working smoothly. These instructions should be similar for any other IDE you might want to use.

In PyCharm, with your project open, click on *Run* from the menu bar and go to *Edit Configurations*. You'll be greeted by a screen similar to this:
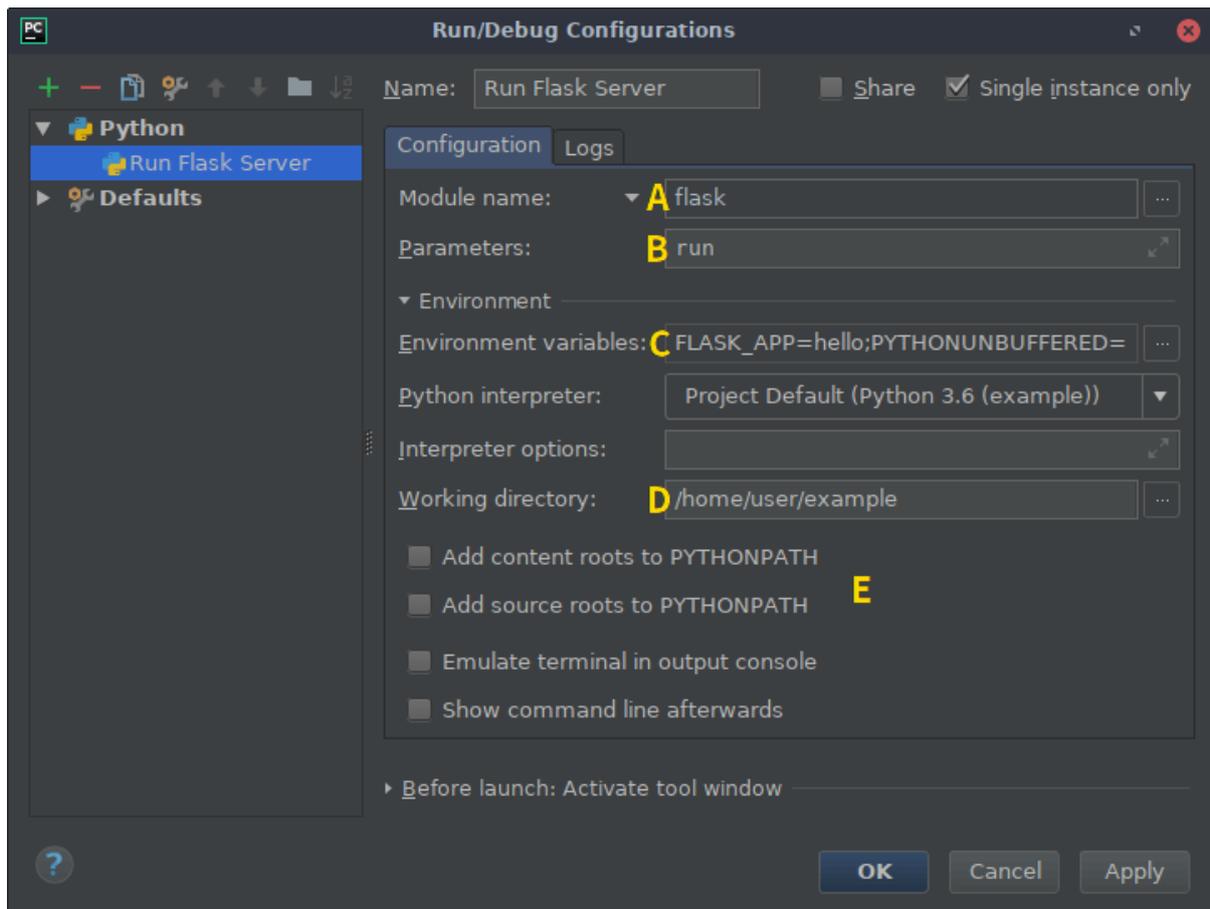
There's quite a few options to change, but once we've done it for one command, we can easily copy the entire configuration and make a single tweak to give us access to other commands, including any custom ones you may implement yourself.

Click the + (*Add New Configuration*) button and select *Python*. Give the configuration a good descriptive name such as "Run Flask Server". For the flask run command, check "Single instance only" since you can't run the server more than once at the same time.

Select *Module name* from the dropdown (**A**) then input flask.

The *Parameters* field (**B**) is set to the CLI command to execute (with any arguments). In this example we use run, which will run the development server.

You can skip this next step if you're using *Environment Variables From dotenv*. We need to add an environment variable (**C**) to identify our application. Click on the browse button and add an entry with FLASK_APP on the left and the Python import or file on the right (hello for example).

Next we need to set the working directory (**D**) to be the folder where our application resides.

If you have installed your project as a package in your virtualenv, you may untick the *PYTHONPATH* options (**E**). This will more accurately match how you deploy the app later.

Click *Apply* to save the configuration, or *OK* to save and close the window. Select the

configuration in the main PyCharm window and click the play button next to it to run the server.

Now that we have a configuration which runs `flask run` from within PyCharm, we can copy that configuration and alter the *Script* argument to run a different CLI command, e.g. `flask shell`.

1.19 Development Server

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

1.19.1 Command Line

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

```
$ export FLASK_APP=my_application
$ export FLASK_ENV=development
$ flask run
```

This enables the development environment, including the interactive debugger and reloader, and then starts the server on *http://localhost:5000/*.

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

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

Note: Prior to Flask 1.0 the `FLASK_ENV` environment variable was not supported and you needed to enable debug mode by exporting `FLASK_DEBUG=1`. This can still be used to control debug mode, but you should prefer setting the development environment as shown above.

1.19.2 In Code

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

Example:

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

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

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

1.20 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.

1.20.1 Command Line Interface

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

For more information see *Command Line Interface*.

1.20.2 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()
```

1.20.3 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).

1.20.4 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 *
```

1.21 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.

1.21.1 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
        ...
```

The *tutorial* is structured this way, see the example code.

Simple Packages

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

You should then end up with something like that:

```
/yourapplication
    /yourapplication
        __init__.py
        /static
            style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

But how do you run your application now? The naive `python  yourapplication/__init__.py` will not work. Let's just say that Python does not want modules in packages to be the startup file. But that is not a big problem, just add a new file called `setup.py` next to the inner `yourapplication` folder with the following contents:

```python
from setuptools import setup

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

In order to run the application you need to export an environment variable that tells Flask where to find the application instance:

```
export FLASK_APP=yourapplication
```

If you are outside of the project directory make sure to provide the exact path to your application directory. Similarly you can turn on the development features like this:

```
export FLASK_ENV=development
```

In order to install and run the application you need to issue the following commands:

```
pip install -e .
flask run
```

What did we gain from this? Now we can restructure the application a bit into multiple modules. The only thing you have to remember is the following quick checklist:

1. the *Flask* application object creation has to be in the `__init__.py` file. That way each module can import it safely and the *__name__* variable will resolve to the correct package.

2. all the view functions (the ones with a `route()` decorator on top) have to be imported in the `__init__.py` file. Not the object itself, but the module it is in. Import the view module **after the application object is created**.

Here's an example `__init__.py`:

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

import yourapplication.views
```

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

```
from yourapplication import app

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

You should then end up with something like that:

```
/yourapplication
    setup.py
    /yourapplication
        __init__.py
        views.py
        /static
            style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

Circular Imports

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

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

Working with Blueprints

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

1.21.2 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 app later.

So why would you want to do this?

1. Testing. You can have instances of the application with different settings to test every case.

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

So how would you then actually implement that?

Basic Factories

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

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

    from yourapplication.model import db
    db.init_app(app)

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

    return app
```

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

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

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

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

Factories & Extensions

It's preferable to create your extensions and app factories so that the extension object does not initially get bound to the application.

Using Flask-SQLAlchemy, as an example, you should not do something along those lines:

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

    db = SQLAlchemy(app)
```

But, rather, in model.py (or equivalent):

```python
db = SQLAlchemy()
```

and in your application.py (or equivalent):

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

    from yourapplication.model import db
    db.init_app(app)
```

Using this design pattern, no application-specific state is stored on the extension object, so one extension object can be used for multiple apps. For more information about the design of extensions refer to *Flask Extension Development*.

Using Applications

To run such an application, you can use the **flask** command:

```
export FLASK_APP=myapp
flask run
```

Flask will automatically detect the factory (create_app or make_app) in myapp. You can also pass arguments to the factory like this:

```
export FLASK_APP="myapp:create_app('dev')"
flask run
```

Then the create_app factory in myapp is called with the string 'dev' as the argument. See *Command Line Interface* for more detail.

Factory Improvements

The factory function above is not very clever, but you can improve it. The following changes are straightforward to implement:

1. Make it possible to pass in configuration values for unit tests 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 being created if necessary.

1.21.3 Application Dispatching

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

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

Working with this Document

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

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

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

In order to use the interactive debugger, debugging must be enabled both on the application and the simple server. Here is the "hello world" example with debugging and `run_simple`:

```
from flask import Flask
from werkzeug.serving import run_simple

app = Flask(__name__)
app.debug = True

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

if __name__ == '__main__':
    run_simple('localhost', 5000, app,
               use_reloader=True, use_debugger=True, use_evalex=True)
```

Combining Applications

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

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

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

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

Dispatch by Subdomain

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

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

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

```python
from threading import Lock

class SubdomainDispatcher(object):

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

    def get_application(self, host):
        host = host.split(':')[0]
```

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

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

This dispatcher can then be used like this:

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

def make_app(subdomain):
    user = get_user_for_subdomain(subdomain)
    if user is None:
        # if there is no user for that subdomain we still have
        # to return a WSGI application that handles that request.
        # We can then just return the NotFound() exception as
        # application which will render a default 404 page.
        # You might also redirect the user to the main page then
        return NotFound()

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

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

Dispatch by Path

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

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

class PathDispatcher(object):

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

```python
    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:

```python
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)
```

1.21.4 Implementing API Exceptions

It's very common to implement RESTful APIs on top of Flask. One of the first things that developers run into is the realization that the builtin exceptions are not expressive enough for APIs and that the content type of text/html they are emitting is not very useful for API consumers.

The better solution than using abort to signal errors for invalid API usage is to implement your own exception type and install an error handler for it that produces the errors in the format the user is expecting.

Simple Exception Class

The basic idea is to introduce a new exception that can take a proper human readable message, a status code for the error and some optional payload to give more context for the error.

This is a simple example:

```
from flask import jsonify

class InvalidUsage(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, payload=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv = dict(self.payload or ())
        rv['message'] = self.message
        return rv
```

A view can now raise that exception with an error message. Additionally some extra payload can be provided as a dictionary through the *payload* parameter.

Registering an Error Handler

At that point views can raise that error, but it would immediately result in an internal server error. The reason for this is that there is no handler registered for this error class. That however is easy to add:

```
@app.errorhandler(InvalidUsage)
def handle_invalid_usage(error):
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response
```

Usage in Views

Here is how a view can use that functionality:

```
@app.route('/foo')
def get_foo():
    raise InvalidUsage('This view is gone', status_code=410)
```

1.21.5 Using URL Processors

New in version 0.7.

Flask 0.7 introduces the concept of URL processors. The idea is that you might have a bunch of resources with common parts in the URL that you don't always explicitly

want to provide. For instance you might have a bunch of URLs that have the language code in it but you don't want to have to handle it in every single function yourself.

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

Internationalized Application URLs

Consider an application like this:

```python
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':

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

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

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

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

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

```
from flask import Flask, g

app = Flask(__name__)

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

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

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


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

Internationalized Blueprint URLs

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

```
from flask import Blueprint, g

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

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

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

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

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

1.21.6 Deploying with Setuptools

Setuptools, is an extension library that is commonly used to distribute Python libraries and extensions. It extends distutils, a basic module installation system shipped with Python to also support various more complex constructs that make larger applications easier to distribute:

- **support for dependencies**: a library or application can declare a list of other libraries it depends on which will be installed automatically for you.

- **package registry**: setuptools registers your package with your Python installation. This makes it possible to query information provided by one package from another package. The best known feature of this system is the entry point support which allows one package to declare an "entry point" that another package can hook into to extend the other package.

- **installation manager**: `pip` can install other libraries for you.

If you have Python 2 (>=2.7.9) or Python 3 (>=3.4) installed from python.org, you will already have pip and setuptools on your system. Otherwise, you will need to install them yourself.

Flask itself, and all the libraries you can find on PyPI are distributed with either setuptools or distutils.

In this case we assume your application is called yourapplication.py and you are not using a module, but a *package*. If you have not yet converted your application into a package, head over to the *Larger Applications* pattern to see how this can be done.

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

Basic Setup Script

Because you have Flask installed, you have setuptools available on your system. Flask already depends upon setuptools.

Standard disclaimer applies: you better use a virtualenv.

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

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

```python
from setuptools import setup

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

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

```python
from setuptools import setup, find_packages

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

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

Tagging Builds

It is useful to distinguish between release and development builds. Add a `setup.cfg` file to configure these options.

```ini
[egg_info]
tag_build = .dev
tag_date = 1

[aliases]
release = egg_info -Db ''
```

Running `python setup.py sdist` will create a development package with ".dev" and the current date appended: `flaskr-1.0.dev20160314.tar.gz`. Running `python setup.py release sdist` will create a release package with only the version: `flaskr-1.0.tar.gz`.

Distributing Resources

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

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

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

Declaring Dependencies

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

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

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

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

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

Installing / Developing

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

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

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

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

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

1.21.7 Deploying with Fabric

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

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

- Fabric 1.0 has to be installed locally. This tutorial assumes the latest version of Fabric.

- The application already has to be a package and requires a working `setup.py` file (*Deploying with Setuptools*).

- In the following example we are using *mod_wsgi* for the remote servers. You can of course use your own favourite server there, but for this example we chose Apache + *mod_wsgi* because it's very easy to setup and has a simple way to reload applications without root access.

Creating the first Fabfile

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

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

```python
from fabric.api import *

# the user to use for the remote commands
```

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

def pack():
    # build the package
    local('python setup.py sdist --formats=gztar', capture=False)

def deploy():
    # figure out the package name and version
    dist = local('python setup.py --fullname', capture=True).strip()
    filename = '%s.tar.gz' % dist

    # upload the package to the temporary folder on the server
    put('dist/%s' % filename, '/tmp/%s' % filename)

    # install the package in the application's virtualenv with pip
    run('/var/www/yourapplication/env/bin/pip install /tmp/%s' % filename)

    # remove the uploaded package
    run('rm -r /tmp/%s' % filename)

    # touch the .wsgi file to trigger a reload in mod_wsgi
    run('touch /var/www/yourapplication.wsgi')
```

Running Fabfiles

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

```
$ fab pack deploy
```

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

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

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

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

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

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

2. Upload a new `application.wsgi` file to the server and the configuration file for the application (eg: `application.cfg`)

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

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

The WSGI File

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

```python
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:

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

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

The Configuration File

Now as mentioned above, the application will find the correct configuration file by looking up the `YOURAPPLICATION_CONFIG` environment variable. So we have to put the configuration in a place where the application will able to find it. Configuration files have the unfriendly quality of being different on all computers, so you do not version them usually.

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

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

First Deployment

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

```
$ fab pack deploy
```

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

Next Steps

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

- Create a *bootstrap* command that initializes new servers. It could initialize a new virtual environment, setup apache appropriately etc.

- Put configuration files into a separate version control repository and symlink the active configs into place.

- You could also put your application code into a repository and check out the latest version on the server and then install. That way you can also easily go back to older versions.

- hook in testing functionality so that you can deploy to an external server and run the test suite.

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

1.21.8 Using SQLite 3 with Flask

In Flask you can easily implement the opening of database connections on demand and closing them when the context dies (usually at the end of the request).

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

```python
import sqlite3
from flask import g

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

def get_db():
    db = getattr(g, '_database', None)
```

```
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

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

Now, to use the database, the application must either have an active application context (which is always true if there is a request in flight) or create an application context itself. At that point the get_db function can be used to get the current database connection. Whenever the context is destroyed the database connection will be terminated.

Note: if you use Flask 0.9 or older you need to use flask._app_ctx_stack.top instead of g as the flask.g object was bound to the request and not application context.

Example:

```
@app.route('/')
def index():
    cur = get_db().cursor()
    ...
```

Note: Please keep in mind that the teardown request and appcontext functions are always executed, even if a before-request handler failed or was never executed. Because of this we have to make sure here that the database is there before we close it.

Connect on Demand

The upside of this approach (connecting on first use) is that this will only open the connection if truly necessary. If you want to use this code outside a request context you can use it in a Python shell by opening the application context by hand:

```
with app.app_context():
    # now you can use get_db()
```

Easy Querying

Now in each request handling function you can access *get_db()* to get the current open database connection. To simplify working with SQLite, a row factory function is useful. It is executed for every result returned from the database to convert the result. For instance, in order to get dictionaries instead of tuples, this could be inserted into the get_db function we created above:

```
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

db.row_factory = make_dicts
```

This will make the sqlite3 module return dicts for this database connection, which are much nicer to deal with. Even more simply, we could place this in get_db instead:

```
db.row_factory = sqlite3.Row
```

This would use Row objects rather than dicts to return the results of queries. These are namedtuple s, so we can access them either by index or by key. For example, assuming we have a sqlite3.Row called r for the rows id, FirstName, LastName, and MiddleInitial:

```
>>> # You can get values based on the row's name
>>> r['FirstName']
John
>>> # Or, you can get them based on index
>>> r[1]
John
# Row objects are also iterable:
>>> for value in r:
...     print(value)
1
John
Doe
M
```

Additionally, it is a good idea to provide a query function that combines getting the cursor, executing and fetching the results:

```
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

This handy little function, in combination with a row factory, makes working with the database much more pleasant than it is by just using the raw cursor and connection objects.

Here is how you can use it:

```
for user in query_db('select * from users'):
    print user['username'], 'has the id', user['user_id']
```

Or if you just want a single result:

```
user = query_db('select * from users where username = ?',
                [the_username], one=True)
if user is None:
    print 'No such user'
else:
    print the_username, 'has the id', user['user_id']
```

To pass variable parts to the SQL statement, use a question mark in the statement and pass in the arguments as a list. Never directly add them to the SQL statement with string formatting because this makes it possible to attack the application using SQL Injections.

Initial Schemas

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

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

You can then create such a database from the Python shell:

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

1.21.9 SQLAlchemy in Flask

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

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

Flask-SQLAlchemy Extension

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

You can download Flask-SQLAlchemy from PyPI.

Declarative

The declarative extension in SQLAlchemy is the most recent method of using SQLAlchemy. It allows you to define tables and models in one go, similar to how Django works. In addition to the following text I recommend the official documentation on the declarative extension.

Here's the example `database.py` module for your application:

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()


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

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

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

```python
from yourapplication.database import db_session

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

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

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


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

```
    email = Column(String(120), unique=True)

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

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

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

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

You can insert entries into the database like this:

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

Querying is simple as well:

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

Manual Object Relational Mapping

Manual object relational mapping has a few upsides and a few downsides versus the declarative approach from above. The main difference is that you define tables and classes separately and map them together. It's more flexible but a little more to type. In general it works like the declarative approach, so make sure to also split up your application into multiple modules in a package.

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

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

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

As in the declarative approach, you need to close the session after each request or application context shutdown. Put this into your application module:

```
from yourapplication.database import db_session

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

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

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

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

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

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

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

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

SQL Abstraction Layer

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

```
from sqlalchemy import create_engine, MetaData, Table

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

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

```
from sqlalchemy import Table

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

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

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

SQLAlchemy will automatically commit for us.

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

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

These results are also dict-like tuples:

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

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

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

For more information about SQLAlchemy, head over to the website.

1.21.10 Uploading Files

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

1. A `<form>` tag is marked with `enctype=multipart/form-data` and an `<input type=file>` is placed in that form.

2. The application accesses the file from the `files` dictionary on the request object.

3. use the `save()` method of the file to save the file permanently somewhere on the filesystem.

A Gentle Introduction

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

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

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

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

So first we need a couple of imports. Most should be straightforward, the `werkzeug.secure_filename()` is explained a little bit later. The `UPLOAD_FOLDER` is where we will store the uploaded files and the `ALLOWED_EXTENSIONS` is the set of allowed file extensions.

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

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

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

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        file = request.files['file']
        # if user does not select file, browser also
        # submit an empty part without filename
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('upload_file',
                                    filename=filename))
    return '''
    <!doctype html>
    <title>Upload new File</title>
    <h1>Upload new File</h1>
    <form method=post enctype=multipart/form-data>
      <input type=file name=file>
      <input type=submit value=Upload>
    </form>
    '''
```

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

Information for the Pros

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

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

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

Now let's look how that function works:

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

Now one last thing is missing: the serving of the uploaded files. In the `upload_file()` we redirect the user to `url_for('uploaded_file', filename=filename)`, that is, `/uploads/filename`. So we write the `uploaded_file()` function to return the file of that name. As of Flask 0.5 we can use a function that does that for us:

```python
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:

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

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

Improving Uploads

New in version 0.6.

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

```python
from flask import Flask, Request

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

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

Connection Reset Issue

When using the local development server, you may get a connection reset error instead of a 413 response. You will get the correct status response when running the app with a production WSGI server.

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

Upload Progress Bars

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

An Easier Solution

Now there are better solutions that work faster and are more reliable. There are JavaScript libraries like jQuery that have form plugins to ease the construction of progress bar.

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

1.21.11 Caching

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

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

Setting up a Cache

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

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

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

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

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

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

Using a Cache

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

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

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

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

Here a full example how this looks like normally:

```python
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
```

1.21.12 View Decorators

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

Login Required Decorator

So let's implement such a decorator. A decorator is a function that wraps and replaces another function. Since the original function is replaced, you need to remember to copy the original function's information to the new function. Use functools.wraps() to handle this for you.

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

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

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

To use the decorator, apply it as innermost decorator to a view function. When applying further decorators, always remember that the route() decorator is the outermost.

```python
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

Note: The `next` value will exist in `request.args` after a `GET` request for the login page. You'll have to pass it along when sending the `POST` request from the login form. You can do this with a hidden input tag, then retrieve it from `request.form` when logging the user in.

```
<input type="hidden" value="{{ request.args.get('next', '') }}"/>
```

Caching Decorator

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

Here is an example cache function. It generates the cache key from a specific prefix (actually a format string) and the current path of the request. Notice that we are using a function that first creates the decorator that then decorates the function. Sounds 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:

```python
from functools import wraps
from flask import request

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

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

Templating Decorator

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

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


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


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

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

Here is the code for that decorator:

```python
from functools import wraps
from flask import request, render_template


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

```
        return decorated_function
    return decorator
```

Endpoint Decorator

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

```python
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"
```

1.21.13 Form Validation with WTForms

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

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

Getting the most out of WTForms with an Extension

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

The Forms

This is an example form for a typical registration page:

```python
from wtforms import Form, BooleanField, StringField, PasswordField, validators

class RegistrationForm(Form):
    username = StringField('Username', [validators.Length(min=4, max=25)])
    email = StringField('Email Address', [validators.Length(min=6, max=35)])
    password = PasswordField('New Password', [
```

```
        validators.DataRequired(),
        validators.EqualTo('confirm', message='Passwords must match')
    ])
    confirm = PasswordField('Repeat Password')
    accept_tos = BooleanField('I accept the TOS', [validators.DataRequired()])
```

In the View

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

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

Notice we're implying that the view is using SQLAlchemy here (*SQLAlchemy in Flask*), but that's not a requirement, of course. Adapt the code as necessary.

Things to remember:

1. create the form from the request `form` value if the data is submitted via the HTTP `POST` method and `args` if the data is submitted as `GET`.

2. to validate the data, call the `validate()` method, which will return `True` if the data validates, `False` otherwise.

3. to access individual values from the form, access *form.<NAME>.data*.

Forms in Templates

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

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

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

```
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
{% endmacro %}
```

This macro accepts a couple of keyword arguments that are forwarded to WTForm's field function, which renders the field for us. The keyword arguments will be inserted as HTML attributes. So, for example, you can call `render_field(form.username, class='username')` to add a class to the input element. Note that WTForms returns standard Python unicode strings, so we have to tell Jinja2 that this data is already HTML-escaped with the `|safe` filter.

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

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

For more information about WTForms, head over to the WTForms website.

1.21.14 Template Inheritance

The most powerful part of Jinja is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

Sounds complicated but is very basic. It's easiest to understand it by starting with an example.

Base Template

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

```
<!doctype html>
<html>
  <head>
```

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

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

Child Template

A child template might look like this:

```
{% extends "layout.html" %}
{% block title %}Index{% endblock %}
{% block head %}
  {{ super() }}
  <style type="text/css">
    .important { color: #336699; }
  </style>
{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome on my awesome homepage.
{% endblock %}
```

The {% extends %} tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent. The extends tag must be the first tag in the template. To render the contents of a block defined in the parent template, use {{ super() }}.

1.21.15 Message Flashing

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

access it next request and only next request. This is usually combined with a layout template that does this. Note that browsers and sometimes web servers enforce a limit on cookie sizes. This means that flashing messages that are too large for session cookies causes message flashing to fail silently.

Simple Flashing

So here is a full example:

```python
from flask import Flask, flash, redirect, render_template, \
     request, url_for

app = Flask(__name__)
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

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

@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != 'admin' or \
                request.form['password'] != 'secret':
            error = 'Invalid credentials'
        else:
            flash('You were successfully logged in')
            return redirect(url_for('index'))
    return render_template('login.html', error=error)
```

And here is the `layout.html` template which does the magic:

```html
<!doctype html>
<title>My Application</title>
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

Here is the `index.html` template which inherits from `layout.html`:

```html
{% extends "layout.html" %}
{% block body %}
```

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

And here is the `login.html` template which also inherits from `layout.html`:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Login</h1>
  {% if error %}
    <p class=error><strong>Error:</strong> {{ error }}
  {% endif %}
  <form method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username value="{{
          request.form.username }}">
      <dt>Password:
      <dd><input type=password name=password>
    </dl>
    <p><input type=submit value=Login>
  </form>
{% endblock %}
```

Flashing With Categories

New in version 0.3.

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

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

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

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

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

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

Filtering Flash Messages

New in version 0.9.

Optionally you can pass a list of categories which filters the results of get_flashed_messages(). This is useful if you wish to render each category in a separate block.

```
{% with errors = get_flashed_messages(category_filter=["error"]) %}
{% if errors %}
<div class="alert-message block-message error">
  <a class="close" href="#">×</a>
  <ul>
    {%- for msg in errors %}
    <li>{{ msg }}</li>
    {% endfor -%}
  </ul>
</div>
{% endif %}
{% endwith %}
```

1.21.16 AJAX with jQuery

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

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

Loading jQuery

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

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

Another method is using Google's AJAX Libraries API to load jQuery:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<script>window.jQuery || document.write('<script src="{{
  url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

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

Where is My Site?

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

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

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

The `|safe` is necessary in Flask before 0.10 so that Jinja does not escape the JSON encoded string with HTML rules. Usually this would be necessary, but we are inside a `script` block here where different rules apply.

Information for Pros

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

In Flask 0.10 it goes a step further and escapes all HTML tags with unicode escapes. This makes it possible for Flask to automatically mark the result as HTML safe.

JSON View Functions

Now let's create a server side function that accepts two URL arguments of numbers which should be added together and then sent back to the application in a JSON object. This is a really ridiculous example and is something you usually would do on the

client side alone, but a simple example that shows how you would use jQuery and Flask nonetheless:

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

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

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

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

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

The HTML

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

```html
<script type=text/javascript>
  $(function() {
    $('a#calculate').bind('click', function() {
      $.getJSON($SCRIPT_ROOT + '/_add_numbers', {
        a: $('input[name="a"]').val(),
        b: $('input[name="b"]').val()
      }, function(data) {
        $("#result").text(data.result);
      });
      return false;
    });
  });
</script>
<h1>jQuery Example</h1>
<p><input type=text size=5 name=a> +
   <input type=text size=5 name=b> =
```

```
   <span id=result>?</span>
<p><a href=# id=calculate>calculate server side</a>
```

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

1. `$(function() { ... })` specifies code that should run once the browser is done loading the basic parts of the page.

2. `$('selector')` selects an element and lets you operate on it.

3. `element.bind('event', func)` specifies a function that should run when the user clicked on the element. If that function returns *false*, the default behavior will not kick in (in this case, navigate to the # URL).

4. `$.getJSON(url, data, func)` sends a GET request to *url* and will send the contents of the *data* object as query parameters. Once the data arrived, it will call the given function with the return value as argument. Note that we can use the *$SCRIPT_ROOT* variable here that we set earlier.

Check out the example source for a full application demonstrating the code on this page, as well as the same thing using `XMLHttpRequest` and `fetch`.

1.21.17 Custom Error Pages

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

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

Common Error Codes

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

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

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

410 Gone Did you know that there the "404 Not Found" has a brother named "410 Gone"? Few people actually implement that, but the idea is that resources that previously existed and got deleted answer with 410 instead of 404. If you are not deleting documents permanently from the database but just mark them as

deleted, do the user a favour and use the 410 code instead and display a message that what they were looking for was deleted for all eternity.

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

Error Handlers

An error handler is a function that returns a response when a type of error is raised, similar to how a view is a function that returns a response when a request URL is matched. It is passed the instance of the error being handled, which is most likely a `HTTPException`. An error handler for "500 Internal Server Error" will be passed uncaught exceptions in addition to explicit 500 errors.

An error handler is registered with the `errorhandler()` decorator or the `register_error_handler()` method. A handler can be registered for a status code, like 404, or for an exception class.

The status code of the response will not be set to the handler's code. Make sure to provide the appropriate HTTP status code when returning a response from a handler.

A handler for "500 Internal Server Error" will not be used when running in debug mode. Instead, the interactive debugger will be shown.

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

```python
from flask import render_template

@app.errorhandler(404)
def page_not_found(e):
    # note that we set the 404 status explicitly
    return render_template('404.html'), 404
```

When using the *application factory pattern*:

```python
from flask import Flask, render_template

def page_not_found(e):
  return render_template('404.html'), 404

def create_app(config_filename):
    app = Flask(__name__)
    app.register_error_handler(404, page_not_found)
    return app
```

An example template might be this:

```html
{% 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 %}
```

1.21.18 Lazily Loading Views

Flask is usually used with the decorators. Decorators are simple and you have the URL right next to the function that is called for that specific URL. However there is a downside to this approach: it means all your code that uses decorators has to be imported upfront or Flask will never actually find your function.

This can be a problem if your application has to import quick. It might have to do that on systems like Google's App Engine or other systems. So if you suddenly notice that your application outgrows this approach you can fall back to a centralized URL mapping.

The system that enables having a central URL map is the add_url_rule() function. Instead of using decorators, you have a file that sets up the application with all URLs.

Converting to Centralized URL Map

Imagine the current application looks somewhat like this:

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

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

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

Then, with the centralized approach you would have one file with the views (views. py) but without any decorator:

```python
def index():
    pass

def user(username):
    pass
```

And then a file that sets up an application which maps the functions to URLs:

```python
from flask import Flask
from yourapplication import views
app = Flask(__name__)
```

```
app.add_url_rule('/', view_func=views.index)
app.add_url_rule('/user/<username>', view_func=views.user)
```

Loading Late

So far we only split up the views and the routing, but the module is still loaded up-front. The trick is to actually load the view function as needed. This can be accomplished with a helper class that behaves just like a function but internally imports the real function on first use:

```python
from werkzeug import import_string, cached_property

class LazyView(object):

    def __init__(self, import_name):
        self.__module__, self.__name__ = import_name.rsplit('.', 1)
        self.import_name = import_name

    @cached_property
    def view(self):
        return import_string(self.import_name)

    def __call__(self, *args, **kwargs):
        return self.view(*args, **kwargs)
```

What's important here is is that *__module__* and *__name__* are properly set. This is used by Flask internally to figure out how to name the URL rules in case you don't provide a name for the rule yourself.

Then you can define your central place to combine the views like this:

```python
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.

```python
def url(import_name, url_rules=[], **options):
    view = LazyView('yourapplication.' + import_name)
    for url_rule in url_rules:
        app.add_url_rule(url_rule, view_func=view, **options)

# add a single route to the index view
```

```
url('views.index', ['/'])

# add two routes to a single function endpoint
url_rules = ['/user/','/user/<username>']
url('views.user', url_rules)
```

One thing to keep in mind is that before and after request handlers have to be in a file that is imported upfront to work properly on the first request. The same goes for any kind of remaining decorator.

1.21.19 MongoKit in Flask

Using a document database rather than a full DBMS gets more common these days. This pattern shows how to use MongoKit, a document mapper library, to integrate with MongoDB.

This pattern requires a running MongoDB server and the MongoKit library installed.

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

Declarative

The default behavior of MongoKit is the declarative one that is based on common ideas from Django or the SQLAlchemy declarative extension.

Here an example app.py module for your application:

```
from flask import Flask
from mongokit import Connection, Document

# configuration
MONGODB_HOST = 'localhost'
MONGODB_PORT = 27017

# create the little application object
app = Flask(__name__)
app.config.from_object(__name__)

# connect to the database
connection = Connection(app.config['MONGODB_HOST'],
                        app.config['MONGODB_PORT'])
```

To define your models, just subclass the *Document* class that is imported from MongoKit. If you've seen the SQLAlchemy pattern you may wonder why we do not have a session and even do not define a *init_db* function here. On the one hand, MongoKit does not have something like a session. This sometimes makes it more to type but also makes it blazingly fast. On the other hand, MongoDB is schemaless. This means

you can modify the data structure from one insert query to the next without any problem. MongoKit is just schemaless too, but implements some validation to ensure data integrity.

Here is an example document (put this also into app.py, e.g.):

```python
from mongokit import ValidationError

def max_length(length):
    def validate(value):
        if len(value) <= length:
            return True
        # must have %s in error format string to have mongokit place key in there
        raise ValidationError('%s must be at most {} characters long'.
 →format(length))
    return validate

class User(Document):
    structure = {
        'name': unicode,
        'email': unicode,
    }
    validators = {
        'name': max_length(50),
        'email': max_length(120)
    }
    use_dot_notation = True
    def __repr__(self):
        return '<User %r>' % (self.name)

# register the User document with our current connection
connection.register([User])
```

This example shows you how to define your schema (named structure), a validator for the maximum character length and uses a special MongoKit feature called *use_dot_notation*. Per default MongoKit behaves like a python dictionary but with *use_dot_notation* set to True you can use your documents like you use models in nearly any other ORM by using dots to separate between attributes.

You can insert entries into the database like this:

```python
>>> from yourapplication.database import connection
>>> from yourapplication.models import User
>>> collection = connection['test'].users
>>> user = collection.User()
>>> user['name'] = u'admin'
>>> user['email'] = u'admin@localhost'
>>> user.save()
```

Note that MongoKit is kinda strict with used column types, you must not use a common *str* type for either *name* or *email* but unicode.

Querying is simple as well:

```
>>> list(collection.User.find())
[<User u'admin'>]
>>> collection.User.find_one({'name': u'admin'})
<User u'admin'>
```

PyMongo Compatibility Layer

If you just want to use PyMongo, you can do that with MongoKit as well. You may use this process if you need the best performance to get. Note that this example does not show how to couple it with Flask, see the above MongoKit code for examples:

```
from MongoKit import Connection

connection = Connection()
```

To insert data you can use the *insert* method. We have to get a collection first, this is somewhat the same as a table in the SQL world.

```
>>> collection = connection['test'].users
>>> user = {'name': u'admin', 'email': u'admin@localhost'}
>>> collection.insert(user)
```

MongoKit will automatically commit for us.

To query your database, you use the collection directly:

```
>>> list(collection.find())
[{u'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u
↪'admin@localhost'}]
>>> collection.find_one({'name': u'admin'})
{u'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u
↪'admin@localhost'}
```

These results are also dict-like objects:

```
>>> r = collection.find_one({'name': u'admin'})
>>> r['email']
u'admin@localhost'
```

For more information about MongoKit, head over to the website.

1.21.20 Adding a favicon

A "favicon" is an icon used by browsers for tabs and bookmarks. This helps to distinguish your website and to give it a unique brand.

A common question is how to add a favicon to a Flask application. First, of course, you need an icon. It should be 16 × 16 pixels and in the ICO file format. This is not a

requirement but a de-facto standard supported by all relevant browsers. Put the icon in your static directory as `favicon.ico`.

Now, to get browsers to find your icon, the correct way is to add a link tag in your HTML. So, for example:

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

That's all you need for most browsers, however some really old ones do not support this standard. The old de-facto standard is to serve this file, with this name, at the website root. If your application is not mounted at the root path of the domain you either need to configure the web server to serve the icon at the root or if you can't do that you're out of luck. If however your application is the root you can simply route a redirect:

```
app.add_url_rule('/favicon.ico',
                 redirect_to=url_for('static', filename='favicon.ico'))
```

If you want to save the extra redirect request you can also write a view using `send_from_directory()`:

```
import os
from flask import send_from_directory

@app.route('/favicon.ico')
def favicon():
    return send_from_directory(os.path.join(app.root_path, 'static'),
                               'favicon.ico', mimetype='image/vnd.microsoft.icon')
```

We can leave out the explicit mimetype and it will be guessed, but we may as well specify it to avoid the extra guessing, as it will always be the same.

The above will serve the icon via your application and if possible it's better to configure your dedicated web server to serve it; refer to the web server's documentation.

See also

- The Favicon article on Wikipedia

1.21.21 Streaming Contents

Sometimes you want to send an enormous amount of data to the client, much more than you want to keep in memory. When you are generating the data on the fly though, how do you send that back to the client without the roundtrip to the filesystem?

The answer is by using generators and direct responses.

Basic Usage

This is a basic view function that generates a lot of CSV data on the fly. The trick is to have an inner function that uses a generator to generate data and to then invoke that function and pass it to a response object:

```python
from flask import Response

@app.route('/large.csv')
def generate_large_csv():
    def generate():
        for row in iter_all_rows():
            yield ','.join(row) + '\n'
    return Response(generate(), mimetype='text/csv')
```

Each `yield` expression is directly sent to the browser. Note though that some WSGI middlewares might break streaming, so be careful there in debug environments with profilers and other things you might have enabled.

Streaming from Templates

The Jinja2 template engine also supports rendering templates piece by piece. This functionality is not directly exposed by Flask because it is quite uncommon, but you can easily do it yourself:

```python
from flask import Response

def stream_template(template_name, **context):
    app.update_template_context(context)
    t = app.jinja_env.get_template(template_name)
    rv = t.stream(context)
    rv.enable_buffering(5)
    return rv

@app.route('/my-large-page.html')
def render_large_template():
    rows = iter_all_rows()
    return Response(stream_template('the_template.html', rows=rows))
```

The trick here is to get the template object from the Jinja2 environment on the application and to call `stream()` instead of `render()` which returns a stream object instead of a string. Since we're bypassing the Flask template render functions and using the template object itself we have to make sure to update the render context ourselves by calling `update_template_context()`. The template is then evaluated as the stream is iterated over. Since each time you do a yield the server will flush the content to the client you might want to buffer up a few items in the template which you can do with `rv.enable_buffering(size)`. 5 is a sane default.

Streaming with Context

New in version 0.9.

Note that when you stream data, the request context is already gone the moment the function executes. Flask 0.9 provides you with a helper that can keep the request context around during the execution of the generator:

```python
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

Without the `stream_with_context()` function you would get a `RuntimeError` at that point.

1.21.22 Deferred Request Callbacks

One of the design principles of Flask is that response objects are created and passed down a chain of potential callbacks that can modify them or replace them. When the request handling starts, there is no response object yet. It is created as necessary either by a view function or by some other component in the system.

What happens if you want to modify the response at a point where the response does not exist yet? A common example for that would be a `before_request()` callback that wants to set a cookie on the response object.

One way is to avoid the situation. Very often that is possible. For instance you can try to move that logic into a `after_request()` callback instead. However, sometimes moving code there makes it more more complicated or awkward to reason about.

As an alternative, you can use `after_this_request()` to register callbacks that will execute after only the current request. This way you can defer code execution from anywhere in the application, based on the current request.

At any time during a request, we can register a function to be called at the end of the request. For example you can remember the current language of the user in a cookie in a `before_request()` callback:

```python
from flask import request, after_this_request

@app.before_request
def detect_user_language():
    language = request.cookies.get('user_lang')

    if language is None:
```

```
        language = guess_language_from_request()

        # when the response exists, set a cookie with the language
        @after_this_request
        def remember_language(response):
            response.set_cookie('user_lang', language)

    g.language = language
```

1.21.23 Adding HTTP Method Overrides

Some HTTP proxies do not support arbitrary HTTP methods or newer HTTP methods (such as PATCH). In that case it's possible to "proxy" HTTP methods through another HTTP method in total violation of the protocol.

The way this works is by letting the client do an HTTP POST request and set the X-HTTP-Method-Override header and set the value to the intended HTTP method (such as PATCH).

This can easily be accomplished with an HTTP middleware:

```python
class HTTPMethodOverrideMiddleware(object):
    allowed_methods = frozenset([
        'GET',
        'HEAD',
        'POST',
        'DELETE',
        'PUT',
        'PATCH',
        'OPTIONS'
    ])
    bodyless_methods = frozenset(['GET', 'HEAD', 'OPTIONS', 'DELETE'])

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        method = environ.get('HTTP_X_HTTP_METHOD_OVERRIDE', '').upper()
        if method in self.allowed_methods:
            method = method.encode('ascii', 'replace')
            environ['REQUEST_METHOD'] = method
        if method in self.bodyless_methods:
            environ['CONTENT_LENGTH'] = '0'
        return self.app(environ, start_response)
```

To use this with Flask this is all that is necessary:

```python
from flask import Flask
```

```
app = Flask(__name__)
app.wsgi_app = HTTPMethodOverrideMiddleware(app.wsgi_app)
```

1.21.24 Request Content Checksums

Various pieces of code can consume the request data and preprocess it. For instance JSON data ends up on the request object already read and processed, form data ends up there as well but goes through a different code path. This seems inconvenient when you want to calculate the checksum of the incoming request data. This is necessary sometimes for some APIs.

Fortunately this is however very simple to change by wrapping the input stream.

The following example calculates the SHA1 checksum of the incoming data as it gets read and stores it in the WSGI environment:

```python
import hashlib

class ChecksumCalcStream(object):

    def __init__(self, stream):
        self._stream = stream
        self._hash = hashlib.sha1()

    def read(self, bytes):
        rv = self._stream.read(bytes)
        self._hash.update(rv)
        return rv

    def readline(self, size_hint):
        rv = self._stream.readline(size_hint)
        self._hash.update(rv)
        return rv

def generate_checksum(request):
    env = request.environ
    stream = ChecksumCalcStream(env['wsgi.input'])
    env['wsgi.input'] = stream
    return stream._hash
```

To use this, all you need to do is to hook the calculating stream in before the request starts consuming data. (Eg: be careful accessing `request.form` or anything of that nature. `before_request_handlers` for instance should be careful not to access it).

Example usage:

```python
@app.route('/special-api', methods=['POST'])
def special_api():
    hash = generate_checksum(request)
    # Accessing this parses the input stream
```

```
    files = request.files
    # At this point the hash is fully constructed.
    checksum = hash.hexdigest()
    return 'Hash was: %s' % checksum
```

1.21.25 Celery Background Tasks

If your application has a long running task, such as processing some uploaded data or sending email, you don't want to wait for it to finish during a request. Instead, use a task queue to send the necessary data to another process that will run the task in the background while the request returns immediately.

Celery is a powerful task queue that can be used for simple background tasks as well as complex multi-stage programs and schedules. This guide will show you how to configure Celery using Flask, but assumes you've already read the First Steps with Celery guide in the Celery documentation.

Install

Celery is a separate Python package. Install it from PyPI using pip:

```
$ pip install celery
```

Configure

The first thing you need is a Celery instance, this is called the celery application. It serves the same purpose as the Flask object in Flask, just for Celery. Since this instance is used as the entry-point for everything you want to do in Celery, like creating tasks and managing workers, it must be possible for other modules to import it.

For instance you can place this in a tasks module. While you can use Celery without any reconfiguration with Flask, it becomes a bit nicer by subclassing tasks and adding support for Flask's application contexts and hooking it up with the Flask configuration.

This is all that is necessary to properly integrate Celery with Flask:

```
from celery import Celery

def make_celery(app):
    celery = Celery(
        app.import_name,
        backend=app.config['CELERY_RESULT_BACKEND'],
        broker=app.config['CELERY_BROKER_URL']
    )
    celery.conf.update(app.config)
```

```
    class ContextTask(celery.Task):
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return self.run(*args, **kwargs)

    celery.Task = ContextTask
    return celery
```

The function creates a new Celery object, configures it with the broker from the application config, updates the rest of the Celery config from the Flask config and then creates a subclass of the task that wraps the task execution in an application context.

An example task

Let's write a task that adds two numbers together and returns the result. We configure Celery's broker and backend to use Redis, create a `celery` application using the factor from above, and then use it to define the task.

```python
from flask import Flask

flask_app = Flask(__name__)
flask_app.config.update(
    CELERY_BROKER_URL='redis://localhost:6379',
    CELERY_RESULT_BACKEND='redis://localhost:6379'
)
celery = make_celery(flask_app)

@celery.task()
def add_together(a, b):
    return a + b
```

This task can now be called in the background:

```python
result = add_together.delay(23, 42)
result.wait()  # 65
```

Run a worker

If you jumped in and already executed the above code you will be disappointed to learn that `.wait()` will never actually return. That's because you also need to run a Celery worker to receive and execute the task.

```
$ celery -A your_application.celery worker
```

The `your_application` string has to point to your application's package or module that creates the `celery` object.

Now that the worker is running, `wait` will return the result once the task is finished.

1.21.26 Subclassing Flask

The `Flask` class is designed for subclassing.

For example, you may want to override how request parameters are handled to preserve their order:

```python
from flask import Flask, Request
from werkzeug.datastructures import ImmutableOrderedMultiDict
class MyRequest(Request):
    """Request subclass to override request parameter storage"""
    parameter_storage_class = ImmutableOrderedMultiDict
class MyFlask(Flask):
    """Flask subclass using the custom request class"""
    request_class = MyRequest
```

This is the recommended approach for overriding or augmenting Flask's internal functionality.

1.22 Deployment Options

While lightweight and easy to use, **Flask's built-in server is not suitable for production** as it doesn't scale well. Some of the options available for properly running Flask in production are documented here.

If you want to deploy your Flask application to a WSGI server not listed here, look up the server documentation about how to use a WSGI app with it. Just remember that your `Flask` application object is the actual WSGI application.

1.22.1 Hosted options

- Deploying Flask on Heroku
- Deploying Flask on OpenShift
- Deploying Flask on Webfaction
- Deploying Flask on Google App Engine
- Deploying Flask on AWS Elastic Beanstalk
- Sharing your Localhost Server with Localtunnel
- Deploying on Azure (IIS)
- Deploying on PythonAnywhere

1.22.2 Self-hosted options

Standalone WSGI Containers

There are popular servers written in Python that contain WSGI applications and serve HTTP. These servers stand alone when they run; you can proxy to them from your web server. Note the section on *Proxy Setups* if you run into issues.

Gunicorn

Gunicorn 'Green Unicorn' is a WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project. It supports both eventlet and greenlet. Running a Flask application on this server is quite simple:

```
gunicorn myproject:app
```

Gunicorn provides many command-line options – see `gunicorn -h`. For example, to run a Flask application with 4 worker processes (`-w 4`) binding to localhost port 4000 (`-b 127.0.0.1:4000`):

```
gunicorn -w 4 -b 127.0.0.1:4000 myproject:app
```

uWSGI

uWSGI is a fast application server written in C. It is very configurable which makes it more complicated to setup than gunicorn.

Running uWSGI HTTP Router:

```
uwsgi --http 127.0.0.1:5000 --module myproject:app
```

For a more optimized setup, see configuring uWSGI and NGINX.

Gevent

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

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

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

Twisted Web

Twisted Web is the web server shipped with Twisted, a mature, non-blocking event-driven networking library. Twisted Web comes with a standard WSGI container which can be controlled from the command line using the `twistd` utility:

```
twistd web --wsgi myproject.app
```

This example will run a Flask application called app from a module named myproject.

Twisted Web supports many flags and options, and the `twistd` utility does as well; see `twistd -h` and `twistd web -h` for more information. For example, to run a Twisted Web server in the foreground, on port 8080, with an application from myproject:

```
twistd -n web --port tcp:8080 --wsgi myproject.app
```

Proxy Setups

If you deploy your application using one of these servers behind an HTTP proxy you will need to rewrite a few headers in order for the application to work. The two problematic values in the WSGI environment usually are REMOTE_ADDR and HTTP_HOST. You can configure your httpd to pass these headers, or you can fix them in middleware. Werkzeug ships a fixer that will solve some common setups, but you might want to write your own WSGI middleware for specific setups.

Here's a simple nginx configuration which proxies to an application served on localhost at port 8000, setting appropriate headers:

```
server {
    listen 80;

    server_name _;

    access_log  /var/log/nginx/access.log;
    error_log   /var/log/nginx/error.log;

    location / {
        proxy_pass         http://127.0.0.1:8000/;
        proxy_redirect     off;

        proxy_set_header   Host                 $host;
        proxy_set_header   X-Real-IP            $remote_addr;
        proxy_set_header   X-Forwarded-For      $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto    $scheme;
    }
}
```

If your httpd is not providing these headers, the most common setup invokes the host being set from X-Forwarded-Host and the remote address from X-Forwarded-For:

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

Trusting Headers

Please keep in mind that it is a security issue to use such a middleware in a non-proxy setup because it will blindly trust the incoming headers which might be forged by malicious clients.

If you want to rewrite the headers from another header, you might want to use a fixer like this:

```
class CustomProxyFix(object):

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        host = environ.get('HTTP_X_FHOST', '')
        if host:
            environ['HTTP_HOST'] = host
        return self.app(environ, start_response)

app.wsgi_app = CustomProxyFix(app.wsgi_app)
```

uWSGI

uWSGI is a deployment option on servers like nginx, lighttpd, and cherokee; see *FastCGI* and *Standalone WSGI Containers* for other options. To use your WSGI application with uWSGI protocol you will need a uWSGI server first. uWSGI is both a protocol and an application server; the application server can serve uWSGI, FastCGI, and HTTP protocols.

The most popular uWSGI server is uwsgi, which we will use for this guide. Make sure to have it installed to follow along.

Watch Out

Please make sure in advance that any app.run() calls you might have in your application file are inside an if __name__ == '__main__': block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to uWSGI.

Starting your app with uwsgi

uwsgi is designed to operate on WSGI callables found in python modules.

Given a flask application in myapp.py, use the following command:

```
$ uwsgi -s /tmp/yourapplication.sock --manage-script-name --mount /
↪yourapplication=myapp:app
```

The `--manage-script-name` will move the handling of `SCRIPT_NAME` to uwsgi, since its smarter about that. It is used together with the `--mount` directive which will make requests to `/yourapplication` be directed to `myapp:app`. If your application is accessible at root level, you can use a single `/` instead of `/yourapplication`. `myapp` refers to the name of the file of your flask application (without extension) or the module which provides app. app is the callable inside of your application (usually the line reads app `= Flask(__name__)`.

If you want to deploy your flask application inside of a virtual environment, you need to also add `--virtualenv /path/to/virtual/environment`. You might also need to add `--plugin python` or `--plugin python3` depending on which python version you use for your project.

Configuring nginx

A basic flask nginx configuration looks like this:

```
location = /yourapplication { rewrite ^ /yourapplication/; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
  include uwsgi_params;
  uwsgi_pass unix:/tmp/yourapplication.sock;
}
```

This configuration binds the application to `/yourapplication`. If you want to have it in the URL root its a bit simpler:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include uwsgi_params;
    uwsgi_pass unix:/tmp/yourapplication.sock;
}
```

mod_wsgi (Apache)

If you are using the Apache webserver, consider using mod_wsgi.

Watch Out

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to mod_wsgi.

Installing *mod_wsgi*

If you don't have *mod_wsgi* installed yet you have to either install it using a package manager or compile it yourself. The mod_wsgi installation instructions cover source installations on UNIX systems.

If you are using Ubuntu/Debian you can apt-get it and activate it as follows:

```
# apt-get install libapache2-mod-wsgi
```

If you are using a yum based distribution (Fedora, OpenSUSE, etc..) you can install it as follows:

```
# yum install mod_wsgi
```

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

```
# pkg install ap22-mod_wsgi2
```

If you are using pkgsrc you can install *mod_wsgi* by compiling the *www/ap2-wsgi* package.

If you encounter segfaulting child processes after the first apache reload you can safely ignore them. Just restart the server.

Creating a *.wsgi* file

To run your application you need a `yourapplication.wsgi` file. This file contains the code *mod_wsgi* is executing on startup to get the application object. The object called *application* in that file is then used as application.

For most applications the following file should be sufficient:

```
from yourapplication import app as application
```

If you don't have a factory function for application creation but a singleton instance you can directly import that one as *application*.

Store that file somewhere that you will find it again (e.g.: /var/www/yourapplication) and make sure that *yourapplication* and all the libraries that are in use are on the python load path. If you don't want to install it system wide consider using a virtual python instance. Keep in mind that you will have to actually install your application into the

virtualenv as well. Alternatively there is the option to just patch the path in the `.wsgi` file before the import:

```python
import sys
sys.path.insert(0, '/path/to/the/application')
```

Configuring Apache

The last thing you have to do is to create an Apache configuration file for your application. In this example we are telling *mod_wsgi* to execute the application under a different user for security reasons:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess yourapplication user=user1 group=group1 threads=5
    WSGIScriptAlias / /var/www/yourapplication/yourapplication.wsgi

    <Directory /var/www/yourapplication>
        WSGIProcessGroup yourapplication
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Note: WSGIDaemonProcess isn't implemented in Windows and Apache will refuse to run with the above configuration. On a Windows system, eliminate those lines:

```
<VirtualHost *>
        ServerName example.com
        WSGIScriptAlias / C:\yourdir\yourapp.wsgi
        <Directory C:\yourdir>
                Order deny,allow
                Allow from all
        </Directory>
</VirtualHost>
```

Note: There have been some changes in access control configuration for Apache 2.4.

Most notably, the syntax for directory permissions has changed from httpd 2.2

```
Order allow,deny
Allow from all
```

to httpd 2.4 syntax

```
Require all granted
```

For more information consult the mod_wsgi documentation.

Troubleshooting

If your application does not run, follow this guide to troubleshoot:

Problem: application does not run, errorlog shows SystemExit ignored You have an `app.run()` call in your application file that is not guarded by an `if __name__ == '__main__':` condition. Either remove that `run()` call from the file and move it into a separate `run.py` file or put it into such an if block.

Problem: application gives permission errors Probably caused by your application running as the wrong user. Make sure the folders the application needs access to have the proper privileges set and the application runs as the correct user (`user` and `group` parameter to the *WSGIDaemonProcess* directive)

Problem: application dies with an error on print Keep in mind that mod_wsgi disallows doing anything with `sys.stdout` and `sys.stderr`. You can disable this protection from the config by setting the *WSGIRestrictStdout* to `off`:

```
WSGIRestrictStdout Off
```

Alternatively you can also replace the standard out in the .wsgi file with a different stream:

```
import sys
sys.stdout = sys.stderr
```

Problem: accessing resources gives IO errors Your application probably is a single .py file you symlinked into the site-packages folder. Please be aware that this does not work, instead you either have to put the folder into the pythonpath the file is stored in, or convert your application into a package.

The reason for this is that for non-installed packages, the module filename is used to locate the resources and for symlinks the wrong filename is picked up.

Support for Automatic Reloading

To help deployment tools you can activate support for automatic reloading. Whenever something changes the `.wsgi` file, *mod_wsgi* will reload all the daemon processes for us.

For that, just add the following directive to your *Directory* section:

```
WSGIScriptReloading On
```

Working with Virtual Environments

Virtual environments have the advantage that they never install the required dependencies system wide so you have a better control over what is used where. If you

want to use a virtual environment with mod_wsgi you have to modify your `.wsgi` file slightly.

Add the following lines to the top of your `.wsgi` file:

```
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

For Python 3 add the following lines to the top of your `.wsgi` file:

```
activate_this = '/path/to/env/bin/activate_this.py'
with open(activate_this) as file_:
    exec(file_.read(), dict(__file__=activate_this))
```

This sets up the load paths according to the settings of the virtual environment. Keep in mind that the path has to be absolute.

FastCGI

FastCGI is a deployment option on servers like nginx, lighttpd, and cherokee; see *uWSGI* and *Standalone WSGI Containers* for other options. To use your WSGI application with any of them you will need a FastCGI server first. The most popular one is flup which we will use for this guide. Make sure to have it installed to follow along.

Watch Out

Please make sure in advance that any `app.run()` calls you might have in your application file are inside an `if __name__ == '__main__':` block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to FastCGI.

Creating a *.fcgi* file

First you need to create the FastCGI server file. Let's call it *yourapplication.fcgi*:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from yourapplication import app

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

This is enough for Apache to work, however nginx and older versions of lighttpd need a socket to be explicitly passed to communicate with the FastCGI server. For that to work you need to pass the path to the socket to the `WSGIServer`:

```
WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()
```

The path has to be the exact same path you define in the server config.

Save the `yourapplication.fcgi` file somewhere you will find it again. It makes sense to have that in `/var/www/yourapplication` or something similar.

Make sure to set the executable bit on that file so that the servers can execute it:

```
# chmod +x /var/www/yourapplication/yourapplication.fcgi
```

Configuring Apache

The example above is good enough for a basic Apache deployment but your *.fcgi* file will appear in your application URL e.g. `example.com/yourapplication.fcgi/news/`. There are few ways to configure your application so that yourapplication.fcgi does not appear in the URL. A preferable way is to use the ScriptAlias and SetHandler configuration directives to route requests to the FastCGI server. The following example uses FastCgiServer to start 5 instances of the application which will handle all incoming requests:

```
LoadModule fastcgi_module /usr/lib64/httpd/modules/mod_fastcgi.so

FastCgiServer /var/www/html/yourapplication/app.fcgi -idle-timeout 300 -processes␣
↪5

<VirtualHost *>
    ServerName webapp1.mydomain.com
    DocumentRoot /var/www/html/yourapplication

    AddHandler fastcgi-script fcgi
    ScriptAlias / /var/www/html/yourapplication/app.fcgi/

    <Location />
        SetHandler fastcgi-script
    </Location>
</VirtualHost>
```

These processes will be managed by Apache. If you're using a standalone FastCGI server, you can use the FastCgiExternalServer directive instead. Note that in the following the path is not real, it's simply used as an identifier to other directives such as AliasMatch:

```
FastCgiServer /var/www/html/yourapplication -host 127.0.0.1:3000
```

If you cannot set ScriptAlias, for example on a shared web host, you can use WSGI middleware to remove yourapplication.fcgi from the URLs. Set .htaccess:

```
<IfModule mod_fcgid.c>
   AddHandler fcgid-script .fcgi
   <Files ~ (\.fcgi)>
       SetHandler fcgid-script
       Options +FollowSymLinks +ExecCGI
   </Files>
</IfModule>

<IfModule mod_rewrite.c>
   Options +FollowSymlinks
   RewriteEngine On
   RewriteBase /
   RewriteCond %{REQUEST_FILENAME} !-f
   RewriteRule ^(.*)$ yourapplication.fcgi/$1 [QSA,L]
</IfModule>
```

Set yourapplication.fcgi:

```python
#!/usr/bin/python
#: optional path to your local python site-packages folder
import sys
sys.path.insert(0, '<your_local_path>/lib/python<your_python_version>/site-
↪packages')

from flup.server.fcgi import WSGIServer
from yourapplication import app

class ScriptNameStripper(object):
   def __init__(self, app):
       self.app = app

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

app = ScriptNameStripper(app)

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

Configuring lighttpd

A basic FastCGI configuration for lighttpd looks like that:

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

```
        "max-procs" => 1
    ))
)

alias.url = (
    "/static/" => "/path/to/your/static/"
)

url.rewrite-once = (
    "^(/static($|/.*))$" => "$1",
    "^(/.*)$" => "/yourapplication.fcgi$1"
)
```

Remember to enable the FastCGI, alias and rewrite modules. This configuration binds the application to /yourapplication. If you want the application to work in the URL root you have to work around a lighttpd bug with the LighttpdCGIRootFix middleware.

Make sure to apply it only if you are mounting the application the URL root. Also, see the Lighty docs for more information on FastCGI and Python (note that explicitly passing a socket to run() is no longer necessary).

Configuring nginx

Installing FastCGI applications on nginx is a bit different because by default no FastCGI parameters are forwarded.

A basic Flask FastCGI configuration for nginx looks like this:

```
location = /yourapplication { rewrite ^ /yourapplication/ last; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
    include fastcgi_params;
    fastcgi_split_path_info ^(/yourapplication)(.*)$;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

This configuration binds the application to /yourapplication. If you want to have it in the URL root it's a bit simpler because you don't have to figure out how to calculate PATH_INFO and SCRIPT_NAME:

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

```
}
```

Running FastCGI Processes

Since nginx and others do not load FastCGI apps, you have to do it by yourself. Super-visor can manage FastCGI processes. You can look around for other FastCGI process managers or write a script to run your *.fcgi* file at boot, e.g. using a SysV `init.d` script. For a temporary solution, you can always run the `.fcgi` script inside GNU screen. See `man screen` for details, and note that this is a manual solution which does not persist across system restart:

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

Debugging

FastCGI deployments tend to be hard to debug on most web servers. Very often the only thing the server log tells you is something along the lines of "premature end of headers". In order to debug the application the only thing that can really give you ideas why it breaks is switching to the correct user and executing the application by hand.

This example assumes your application is called *application.fcgi* and that your web server user is *www-data*:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcgi", line 4, in <module>
ImportError: No module named yourapplication
```

In this case the error seems to be "yourapplication" not being on the python path. Common problems are:

- Relative paths being used. Don't rely on the current working directory.

- The code depending on environment variables that are not set by the web server.

- Different python interpreters being used.

CGI

If all other deployment methods do not work, CGI will work for sure. CGI is sup-ported by all major servers but usually has a sub-optimal performance.

This is also the way you can use a Flask application on Google's App Engine, where execution happens in a CGI-like environment.

Watch Out

Please make sure in advance that any app.run() calls you might have in your application file are inside an if __name__ == '__main__': block or moved to a separate file. Just make sure it's not called because this will always start a local WSGI server which we do not want if we deploy that application to CGI / app engine.

With CGI, you will also have to make sure that your code does not contain any print statements, or that sys.stdout is overridden by something that doesn't write into the HTTP response.

Creating a *.cgi* file

First you need to create the CGI application file. Let's call it yourapplication.cgi:

```python
#!/usr/bin/python
from wsgiref.handlers import CGIHandler
from yourapplication import app

CGIHandler().run(app)
```

Server Setup

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

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

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

On shared webhosting, though, you might not have access to your Apache config. In this case, a file called .htaccess, sitting in the public directory you want your app to be available, works too but the ScriptAlias directive won't work in that case:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f # Don't interfere with static files
RewriteRule ^(.*)$ /path/to/the/application.cgi/$1 [L]
```

For more information consult the documentation of your webserver.

1.23 Becoming Big

Here are your options when growing your codebase or scaling your application.

1.23.1 Read the Source.

Flask started in part to demonstrate how to build your own framework on top of existing well-used tools Werkzeug (WSGI) and Jinja (templating), and as it developed, it became useful to a wide audience. As you grow your codebase, don't just use Flask – understand it. Read the source. Flask's code is written to be read; its documentation is published so you can use its internal APIs. Flask sticks to documented APIs in upstream libraries, and documents its internal utilities so that you can find the hook points needed for your project.

1.23.2 Hook. Extend.

The *API* docs are full of available overrides, hook points, and *Signals*. You can provide custom classes for things like the request and response objects. Dig deeper on the APIs you use, and look for the customizations which are available out of the box in a Flask release. Look for ways in which your project can be refactored into a collection of utilities and Flask extensions. Explore the many extensions in the community, and look for patterns to build your own extensions if you do not find the tools you need.

1.23.3 Subclass.

The Flask class has many methods designed for subclassing. You can quickly add or customize behavior by subclassing Flask (see the linked method docs) and using that subclass wherever you instantiate an application class. This works well with *Application Factories*. See *Subclassing Flask* for an example.

1.23.4 Wrap with middleware.

The *Application Dispatching* chapter shows in detail how to apply middleware. You can introduce WSGI middleware to wrap your Flask instances and introduce fixes and changes at the layer between your Flask application and your HTTP server. Werkzeug includes several middlewares.

1.23.5 Fork.

If none of the above options work, fork Flask. The majority of code of Flask is within Werkzeug and Jinja2. These libraries do the majority of the work. Flask is just the paste that glues those together. For every project there is the point where the underlying framework gets in the way (due to assumptions the original developers had). This is

natural because if this would not be the case, the framework would be a very complex system to begin with which causes a steep learning curve and a lot of user frustration.

This is not unique to Flask. Many people use patched and modified versions of their framework to counter shortcomings. This idea is also reflected in the license of Flask. You don't have to contribute any changes back if you decide to modify the framework.

The downside of forking is of course that Flask extensions will most likely break because the new framework has a different import name. Furthermore integrating upstream changes can be a complex process, depending on the number of changes. Because of that, forking should be the very last resort.

1.23.6 Scale like a pro.

For many web applications the complexity of the code is less an issue than the scaling for the number of users or data entries expected. Flask by itself is only limited in terms of scaling by your application code, the data store you want to use and the Python implementation and webserver you are running on.

Scaling well means for example that if you double the amount of servers you get about twice the performance. Scaling bad means that if you add a new server the application won't perform any better or would not even support a second server.

There is only one limiting factor regarding scaling in Flask which are the context local proxies. They depend on context which in Flask is defined as being either a thread, process or greenlet. If your server uses some kind of concurrency that is not based on threads or greenlets, Flask will no longer be able to support these global proxies. However the majority of servers are using either threads, greenlets or separate processes to achieve concurrency which are all methods well supported by the underlying Werkzeug library.

1.23.7 Discuss with the community.

The Flask developers keep the framework accessible to users with codebases big and small. If you find an obstacle in your way, caused by Flask, don't hesitate to contact the developers on the mailinglist or IRC channel. The best way for the Flask and Flask extension developers to improve the tools for larger applications is getting feedback from users.

API Reference

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

2.1 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.

2.1.1 Application Object

class flask.**Flask**(*import_name*, *static_url_path=None*, *static_folder='static'*, *static_host=None*, *host_matching=False*, *subdomain_matching=False*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=None*)

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see open_resource().

Usually you create a Flask instance in your main module or in the __init__.py file of your package like this:

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

About the First Parameter

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, *__name__* is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in yourapplication/app.py you should create it with one of the two versions below:

```python
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with *__name__*, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

New in version 0.7: The *static_url_path*, *static_folder*, and *template_folder* parameters were added.

New in version 0.8: The *instance_path* and *instance_relative_config* parameters were added.

New in version 0.11: The *root_path* parameter was added.

New in version 1.0: The host_matching and static_host parameters were added.

New in version 1.0: The subdomain_matching parameter was added. Subdomain matching needs to be enabled manually now. Setting SERVER_NAME does not implicitly enable it.

> **Parameters**
>
> - **import_name** – the name of the application package
> - **static_url_path** – can be used to specify a different path for the static files on the web. Defaults to the name of the

static_folder folder.

- **static_folder** – the folder with static files that should be served at *static_url_path*. Defaults to the `'static'` folder in the root path of the application.

- **static_host** – the host to use when adding the static route. Defaults to None. Required when using `host_matching=True` with a `static_folder` configured.

- **host_matching** – set `url_map.host_matching` attribute. Defaults to False.

- **subdomain_matching** – consider the subdomain relative to `SERVER_NAME` when matching routes. Defaults to False.

- **template_folder** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.

- **instance_path** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.

- **instance_relative_config** – if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.

- **root_path** – Flask by default will automatically calculate the path to the root of the application. In certain situations this cannot be achieved (for instance if the package is a Python 3 namespace package) and needs to be manually defined.

add_template_filter(*f, name=None*)
    Register a custom template filter. Works exactly like the `template_filter()` decorator.

        Parameters name – the optional name of the filter, otherwise the function name will be used.

add_template_global(*f, name=None*)
    Register a custom template global function. Works exactly like the `template_global()` decorator.

    New in version 0.10.

        Parameters name – the optional name of the global function, otherwise the function name will be used.

add_template_test(*f, name=None*)
    Register a custom template test. Works exactly like the `template_test()` decorator.

    New in version 0.10.

> **Parameters name** – the optional name of the test, otherwise the function name will be used.

add_url_rule(*rule, endpoint=None, view_func=None, provide_automatic_options=None, **options*)

Connects a URL rule. Works exactly like the `route()` decorator. If a view_func is provided it will be registered with the endpoint.

Basically this example:

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

Is equivalent to the following:

```python
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:

```python
app.view_functions['index'] = index
```

Internally `route()` invokes `add_url_rule()` so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to *URL Route Registrations*.

Changed in version 0.2: *view_func* parameter added.

Changed in version 0.6: `OPTIONS` is added automatically as method.

> **Parameters**
>
> - **rule** – the URL rule as string
>
> - **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
>
> - **view_func** – the function to call when serving a request to the provided endpoint
>
> - **provide_automatic_options** – controls whether the `OPTIONS` method should be added automatically. This can also be controlled by setting the `view_func.provide_automatic_options = False` before adding the rule.
>
> - **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

after_request(*f*)

Register a function to be run after each request.

Your function must take one parameter, an instance of `response_class` and return a new response object or the same (see `process_response()`).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

after_request_funcs = None

A dictionary with lists of functions that should be called after each request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to close database connections. To register a function here, use the `after_request()` decorator.

app_context()

Create an `AppContext`. Use as a `with` block to push the context, which will make `current_app` point at this application.

An application context is automatically pushed by `RequestContext.push()` when handling a request, and when running a CLI command. Use this to manually create a context outside of these situations.

```
with app.app_context():
    init_db()
```

See *The Application Context*.

New in version 0.9.

app_ctx_globals_class

The class that is used for the g instance.

Example use cases for a custom class:

1. Store arbitrary attributes on flask.g.

2. Add a property for lazy per-request database connectors.

3. Return None instead of AttributeError on unexpected attributes.

4. Raise exception if an unexpected attr is set, a "controlled" flask.g.

In Flask 0.9 this property was called *request_globals_class* but it was changed in 0.10 to `app_ctx_globals_class` because the flask.g object is now application context scoped.

New in version 0.10.

alias of `_AppCtxGlobals`

auto_find_instance_path()

Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

New in version 0.8.

before_first_request(*f*)
> Registers a function to be run before the first request to this instance of the application.

> The function will be called without any arguments and its return value is ignored.

> New in version 0.8.

before_first_request_funcs = None
> A list of functions that will be called at the beginning of the first request to this instance. To register a function, use the before_first_request() decorator.

> New in version 0.8.

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

> For example, this can be used to open a database connection, or to load the logged in user from the session.

> The function will be called without any arguments. If it returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

before_request_funcs = None
> A dictionary with lists of functions that will be called at the beginning of each request. The key of the dictionary is the name of the blueprint this function is active for, or None for all requests. To register a function, use the before_request() decorator.

blueprints = None
> all the attached blueprints in a dictionary 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.

cli = None
> The click command line context for this application. Commands registered here show up in the **flask** command once the application has been discovered. The default commands are provided by Flask itself and can be overridden.

> This is an instance of a click.Group object.

config = None
> The configuration dictionary as Config. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

config_class
> The class that is used for the config attribute of this app. Defaults to Config.

> Example use cases for a custom class:

1. Default values for certain config options.

2. Access to config values through attributes in addition to keys.

New in version 0.11.

alias of `Config`

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

create_global_jinja_loader()
> Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the `jinja_loader()` function instead.
>
> The global loader dispatches between the loaders of the application and the individual blueprints.
>
> New in version 0.7.

create_jinja_environment()
> Creates the Jinja2 environment based on `jinja_options` and `select_jinja_autoescape()`. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.
>
> New in version 0.5.
>
> Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

create_url_adapter(*request*)
> Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly.
>
> New in version 0.6.
>
> Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.
>
> Changed in version 1.0: `SERVER_NAME` no longer implicitly enables subdomain matching. Use `subdomain_matching` instead.

debug
> Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. This maps to the `DEBUG` config key. This is enabled when env is `'development'` and is overridden by the `FLASK_DEBUG` environment variable. It may not behave as expected if set in code.
>
> **Do not enable debug mode when deploying in production.**
>
> Default: `True` if env is `'development'`, or `False` otherwise.

default_config = ImmutableDict({'JSON_AS_ASCII': True, 'USE_X_SENDFILE': False, 'SE
    Default configuration parameters.

dispatch_request()
    Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call `make_response()`.

    Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

do_teardown_appcontext(*exc=<object object>*)
    Called right before the application context is popped.

    When handling a request, the application context is popped after the request context. See `do_teardown_request()`.

    This calls all functions decorated with `teardown_appcontext()`. Then the `appcontext_tearing_down` signal is sent.

    This is called by `AppContext.pop()`.

    New in version 0.9.

do_teardown_request(*exc=<object object>*)
    Called after the request is dispatched and the response is returned, right before the request context is popped.

    This calls all functions decorated with `teardown_request()`, and `Blueprint.teardown_request()` if a blueprint handled the request. Finally, the `request_tearing_down` signal is sent.

    This is called by `RequestContext.pop()`, which may be delayed during testing to maintain access to resources.

        Parameters exc – An unhandled exception raised while dispatching the request. Detected from the current exception information if not passed. Passed to each teardown function.

    Changed in version 0.9: Added the exc argument.

endpoint(*endpoint*)
    A decorator to register a function as an endpoint. Example:

```python
@app.endpoint('example.endpoint')
def example():
    return "example"
```

        Parameters endpoint – the name of the endpoint

env
    What environment the app is running in. Flask and extensions may enable behaviors based on the environment, such as enabling debug mode. This

maps to the ENV config key. This is set by the FLASK_ENV environment variable and may not behave as expected if set in code.

Do not enable development when deploying in production.

Default: 'production'

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 an error handler, use the errorhandler() decorator.

errorhandler(*code_or_exception*)

Register a function to handle errors by code or exception class.

A decorator that is used to register a function given an error code. Example:

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

You can also register handlers for arbitrary exceptions:

```python
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

New in version 0.7: Use register_error_handler() instead of modifying error_handler_spec directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the HTTPException class.

> **Parameters** **code_or_exception** – the code as integer for the handler, or an arbitrary exception

extensions = None

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things. For backwards compatibility extensions should register themselves like this:

```python
if not hasattr(app, 'extensions'):
    app.extensions = {}
app.extensions['extensionname'] = SomeObject()
```

The key must match the name of the extension module. For example in case of a "Flask-Foo" extension in *flask_foo*, the key would be 'foo'.

New in version 0.7.

full_dispatch_request()

Dispatches the request and on top of that performs request pre and post-processing as well as HTTP exception catching and error handling.

New in version 0.7.

get_send_file_max_age(*filename*)

Provides default cache_timeout for the send_file() functions.

By default, this function returns SEND_FILE_MAX_AGE_DEFAULT from the configuration of current_app.

Static file functions such as send_from_directory() use this function, and send_file() calls this function on current_app when the given cache_timeout is None. If a cache_timeout is given in send_file(), that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```python
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

New in version 0.9.

got_first_request

This attribute is set to True if the application started handling the first request.

New in version 0.8.

handle_exception(*e*)

Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

New in version 0.3.

handle_http_exception(*e*)

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

New in version 0.3.

handle_url_build_error(*error, endpoint, values*)

Handle BuildError on url_for().

handle_user_exception(*e*)

This method is called whenever an exception occurs that should be handled.

A special case are HTTPExceptions which are forwarded by this function to the `handle_http_exception()` method. This function will either return a response value or reraise the exception with the same traceback.

Changed in version 1.0: Key errors raised from request data like `form` show the the bad key in debug mode rather than a generic bad request message.

New in version 0.7.

has_static_folder
This is `True` if the package bound object's container has a folder for static files.

New in version 0.5.

import_name = None
The name of the package or module that this app belongs to. Do not change this once it is set by the constructor.

inject_url_defaults(*endpoint*, *values*)
Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

New in version 0.7.

instance_path = None
Holds the path to the instance folder.

New in version 0.8.

iter_blueprints()
Iterates over all blueprints by the order they were registered.

New in version 0.11.

jinja_env
The Jinja2 environment used to load templates.

jinja_environment
The class that is used for the Jinja environment.

New in version 0.11.

alias of `Environment`

jinja_loader
The Jinja loader for this package bound object.

New in version 0.5.

jinja_options = ImmutableDict({'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.
Options that are passed directly to the Jinja2 environment.

json_decoder
The JSON decoder class to use. Defaults to `JSONDecoder`.

New in version 0.10.

alias of `JSONDecoder`

json_encoder
> The JSON encoder class to use. Defaults to `JSONEncoder`.
>
> New in version 0.10.
>
> alias of `JSONEncoder`

log_exception(*exc_info*)
> Logs an exception. This is called by `handle_exception()` if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the `logger`.
>
> New in version 0.8.

logger
> The `'flask.app'` logger, a standard Python `Logger`.
>
> In debug mode, the logger's `level` will be set to `DEBUG`.
>
> If there are no handlers configured, a default handler will be added. See *Logging* for more information.
>
> Changed in version 1.0: Behavior was simplified. The logger is always named `flask.app`. The level is only set during configuration, it doesn't check `app.debug` each time. Only one format is used, not different ones depending on `app.debug`. No handlers are removed, and a handler is only added if no handlers are already configured.
>
> New in version 0.3.

make_config(*instance_relative=False*)
> Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.
>
> New in version 0.8.

make_default_options_response()
> This method is called to create the default `OPTIONS` response. This can be changed through subclassing to change the default behavior of `OPTIONS` responses.
>
> New in version 0.7.

make_null_session()
> Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the `session_interface`.
>
> New in version 0.7.

make_response(*rv*)
> Convert the return value from a view function to an instance of `response_class`.

Parameters `rv` – the return value from the view function. The view function must return a response. Returning None, or the view ending without returning, is not allowed. The following types are allowed for `view_rv`:

str (unicode in Python 2) A response object is created with the string encoded to UTF-8 as the body.

bytes (`str` in Python 2) A response object is created with the bytes as the body.

tuple Either (`body,  status,  headers`), (`body,  status`), or (`body,  headers`), where body is any of the other types allowed here, `status` is a string or an integer, and `headers` is a dictionary or a list of (`key,  value`) tuples. If body is a `response_class` instance, `status` overwrites the exiting value and `headers` are extended.

response_class The object is returned unchanged.

other `Response` class The object is coerced to `response_class`.

callable() The function is called as a WSGI application. The result is used to create a response object.

Changed in version 0.9: Previously a tuple was interpreted as the arguments for the response object.

make_shell_context()
Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

New in version 0.11.

name
The name of the application. This is usually the import name with the difference that it's guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

New in version 0.8.

open_instance_resource(*resource, mode='rb'*)
Opens a resource from the application's instance folder (`instance_path`). Otherwise works like `open_resource()`. Instance resources can also be opened for writing.

Parameters

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.

- **mode** – resource file opening mode, default is 'rb'.

open_resource(*resource, mode='rb'*)
Opens a resource from the application's resource folder. To see how this

works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the schema.sql file you would do the following:

```python
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

> **Parameters**
>
> - **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
>
> - **mode** – resource file opening mode, default is 'rb'.

open_session(*request*)

Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the secret_key is set. Instead of overriding this method we recommend replacing the session_interface.

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

permanent_session_lifetime

A timedelta which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the PERMANENT_SESSION_LIFETIME configuration key. Defaults to timedelta(days=31)

preprocess_request()

Called before the request is dispatched. Calls url_value_preprocessors registered with the app and the current blueprint (if any). Then calls before_request_funcs registered with the app and the blueprint.

If any before_request() handler returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

preserve_context_on_exception

Returns the value of the PRESERVE_CONTEXT_ON_EXCEPTION configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

process_response(*response*)
> Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the after_request() decorated functions.
>
> Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.
>
> > **Parameters response** – a response_class object.
> >
> > **Returns** a new response object or the same, has to be an instance of response_class.

propagate_exceptions
> Returns the value of the PROPAGATE_EXCEPTIONS configuration value in case it's set, otherwise a sensible default is returned.
>
> New in version 0.7.

register_blueprint(*blueprint*, ***options*)
> Register a Blueprint on the application. Keyword arguments passed to this method will override the defaults set on the blueprint.
>
> Calls the blueprint's register() method after recording the blueprint in the application's blueprints.
>
> > **Parameters**
> >
> > - **blueprint** – The blueprint to register.
> > - **url_prefix** – Blueprint routes will be prefixed with this.
> > - **subdomain** – Blueprint routes will match on this subdomain.
> > - **url_defaults** – Blueprint routes will use these default values for view arguments.
> > - **options** – Additional keyword arguments are passed to BlueprintSetupState. They can be accessed in record() callbacks.
>
> New in version 0.7.

register_error_handler(*code_or_exception*, *f*)
> Alternative error attach function to the errorhandler() decorator that is more straightforward to use for non decorator usage.
>
> New in version 0.7.

request_class
> The class that is used for request objects. See Request for more information.
>
> alias of Request

request_context(*environ*)
> Create a RequestContext representing a WSGI environment. Use a with block to push the context, which will make request point at this request.

See *The Request Context*.

Typically you should not call this from your own code. A request context is automatically pushed by the `wsgi_app()` when handling a request. Use `test_request_context()` to create an environment and context instead of this method.

> **Parameters** `environ` – a WSGI environment

response_class
> The class that is used for response objects. See `Response` for more information.

> alias of `Response`

root_path = None
> Absolute path to the package on the filesystem. Used to look up resources contained in the package.

route(*rule*, ***options*)
> A decorator that is used to register a view function for a given URL rule. This does the same thing as `add_url_rule()` but is intended for decorator usage:

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

> For more information refer to *URL Route Registrations*.

> **Parameters**
>
> - **rule** – the URL rule as string
>
> - **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
>
> - **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (GET, POST etc.). By default a rule just listens for GET (and implicitly HEAD). Starting with Flask 0.6, OPTIONS is implicitly added and handled by the standard request handling.

run(*host=None*, *port=None*, *debug=None*, *load_dotenv=True*, ***options*)
> Runs the application on a local development server.

> Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see *Deployment Options* for WSGI server recommendations.

> If the debug flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass use_evalex=False as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **flask** command line script's run support.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke run() with debug=True and use_reloader=False. Setting use_debugger to True without being in debug mode won't catch any exceptions because there won't be any to catch.

Parameters

- **host** – the hostname to listen on. Set this to '0.0.0.0' to have the server available externally as well. Defaults to '127.0.0. 1' or the host in the SERVER_NAME config variable if present.

- **port** – the port of the webserver. Defaults to 5000 or the port defined in the SERVER_NAME config variable if present.

- **debug** – if given, enable or disable debug mode. See debug.

- **load_dotenv** – Load the nearest .env and .flaskenv files to set environment variables. Will also change the working directory to the directory containing the first file found.

- **options** – the options to be forwarded to the underlying Werkzeug server. See werkzeug.serving.run_simple() for more information.

Changed in version 1.0: If installed, python-dotenv will be used to load environment variables from .env and .flaskenv files.

If set, the FLASK_ENV and FLASK_DEBUG environment variables will override env and debug.

Threaded mode is enabled by default.

Changed in version 0.10: The default port is now picked from the SERVER_NAME variable.

save_session(*session*, *response*)
Saves the session if it needs updates. For the default implementation, check open_session(). Instead of overriding this method we recommend replacing the session_interface.

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

secret_key
> If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.
>
> This attribute can also be configured from the config with the `SECRET_KEY` configuration key. Defaults to `None`.

select_jinja_autoescape(*filename*)
> Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.
>
> New in version 0.5.

send_file_max_age_default
> A `timedelta` which is used as default cache_timeout for the `send_file()` functions. The default is 12 hours.
>
> This attribute can also be configured from the config with the `SEND_FILE_MAX_AGE_DEFAULT` configuration key. This configuration variable can also be set with an integer value used as seconds. Defaults to `timedelta(hours=12)`

send_static_file(*filename*)
> Function used internally to send static files from the static folder to the browser.
>
> New in version 0.5.

session_cookie_name
> The secure cookie uses this for the name of the session cookie.
>
> This attribute can also be configured from the config with the `SESSION_COOKIE_NAME` configuration key. Defaults to `'session'`

session_interface = <flask.sessions.SecureCookieSessionInterface object>
> the session interface to use. By default an instance of `SecureCookieSessionInterface` is used here.
>
> New in version 0.8.

shell_context_processor(*f*)
> Registers a shell context processor function.
>
> New in version 0.11.

shell_context_processors = None
> A list of shell context processor functions that should be run when a shell context is created.
>
> New in version 0.11.

should_ignore_error(*error*)
    This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

    New in version 0.10.

static_folder
    The absolute path to the configured static folder.

static_url_path
    The URL prefix that the static route will be registered for.

teardown_appcontext(*f*)
    Registers a function to be called when the application context ends. These functions are typically also called when the request context is popped.

    Example:

```
ctx = app.app_context()
ctx.push()
...
ctx.pop()
```

    When `ctx.pop()` is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

    Since a request context typically also manages an application context it would also be called when you pop a request context.

    When a teardown function was called because of an unhandled exception it will be passed an error object. If an `errorhandler()` is registered, it will handle the exception and the teardown will not receive it.

    The return values of teardown functions are ignored.

    New in version 0.9.

teardown_appcontext_funcs = None
    A list of functions that are called when the application context is destroyed. Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

    New in version 0.9.

teardown_request(*f*)
    Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

    Example:

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

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

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of an exception it will be passed an error object.

The return values of teardown functions are ignored.

Debug Note

In debug mode Flask will not tear down a request on an exception immediately. Instead it will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable.

---

teardown_request_funcs = None

> A dictionary with lists of functions that are called after each request, even if an exception has occurred. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. These functions are not allowed to modify the request, and their return values are ignored. If an exception occurred while processing the request, it gets passed to each teardown_request function. To register a function here, use the `teardown_request()` decorator.
>
> New in version 0.7.

template_context_processors = None

> A dictionary with list of functions that are called without argument to populate the template context. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. Each returns a dictionary that the template context is updated with. To register a function here, use the `context_processor()` decorator.

template_filter(*name=None*)

> A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

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

Parameters name – the optional name of the filter, otherwise the function name will be used.

template_folder = None

Location of the template files to be added to the template lookup. `None` if templates should not be added.

template_global(*name=None*)

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

New in version 0.10.

Parameters name – the optional name of the global function, otherwise the function name will be used.

template_test(*name=None*)

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

New in version 0.10.

Parameters name – the optional name of the test, otherwise the function name will be used.

templates_auto_reload

Reload templates when they are changed. Used by `create_jinja_environment()`.

This attribute can be configured with `TEMPLATES_AUTO_RELOAD`. If not set, it will be enabled in debug mode.

New in version 1.0: This property was added but the underlying config and behavior already existed.

test_cli_runner(***kwargs*)

Create a CLI runner for testing CLI commands. See *Testing CLI Commands*.

Returns an instance of `test_cli_runner_class`, by default `FlaskCliRunner`. The Flask app object is passed as the first argument.

New in version 1.0.

test_cli_runner_class = None

The `CliRunner` subclass, by default `FlaskCliRunner` that is used by `test_cli_runner()`. Its `__init__` method should take a Flask app object as the first argument.

New in version 1.0.

test_client(*use_cookies=True, **kwargs*)

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

Note that if you are testing for assertions or exceptions in your application code, you must set `app.testing = True` in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the `testing` attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a `with` block to defer the closing down of the context until the end of the `with` block. This is useful if you want to access the context locals for testing:

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

Additionally, you may pass optional keyword arguments that will then be passed to the application's `test_client_class` constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
    def __init__(self, *args, **kwargs):
        self._authentication = kwargs.pop("authentication")
        super(CustomClient,self).__init__( *args, **kwargs)

app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See `FlaskClient` for more information.

Changed in version 0.4: added support for `with` block usage for the client.

New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the `test_client_class` attribute.

Changed in version 0.11: Added **kwargs* to support passing additional keyword arguments to the constructor of `test_client_class`.

test_client_class = None
> the test client that is used with when *test_client* is used.

New in version 0.7.

test_request_context(**args*, ***kwargs*)
> Create a `RequestContext` for a WSGI environment created from the given values. This is mostly useful during testing, where you may want to run a function that uses request data without dispatching a full request.
>
> See *The Request Context*.
>
> Use a `with` block to push the context, which will make `request` point at the request for the created environment.

```
with test_request_context(...):
    generate_report()
```

> When using the shell, it may be easier to push and pop the context manually to avoid indentation.

```
ctx = app.test_request_context(...)
ctx.push()
...
ctx.pop()
```

> Takes the same arguments as Werkzeug's `EnvironBuilder`, with some defaults from the application. See the linked Werkzeug docs for most of the available arguments. Flask-specific behavior is listed here.
>
> > **Parameters**
> >
> > - **path** – URL path being requested.
> > - **base_url** – Base URL where the app is being served, which `path` is relative to. If not given, built from `PREFERRED_URL_SCHEME`, `subdomain`, `SERVER_NAME`, and `APPLICATION_ROOT`.
> > - **subdomain** – Subdomain name to append to `SERVER_NAME`.
> > - **url_scheme** – Scheme to use instead of `PREFERRED_URL_SCHEME`.
> > - **data** – The request body, either as a string or a dict of form keys and values.
> > - **json** – If given, this is serialized as JSON and passed as data. Also defaults `content_type` to `application/json`.
> > - **args** – other positional arguments passed to `EnvironBuilder`.

> - **kwargs** – other keyword arguments passed to `EnvironBuilder`.

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 test 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.
>
> Changed in version 1.0: Bad request errors are not trapped by default in debug mode.
>
> New in version 0.8.

update_template_context(*context*)

> Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.
>
> > **Parameters** **context** – the context as a dictionary that is updated in place to add extra variables.

url_build_error_handlers = None

> A list of functions that are called when `url_for()` raises a `BuildError`. Each function registered here is called with *error*, *endpoint* and *values*. If a function returns `None` or raises a `BuildError` the next function is tried.
>
> New in version 0.9.

url_default_functions = None

> A dictionary with lists of functions that can be used as URL value preprocessors. The key `None` here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary of URL values before they are used as the keyword arguments of the view function. For each function registered this one

should also provide a `url_defaults()` function that adds the parameters automatically again that were removed that way.

New in version 0.7.

url_defaults(*f*)

Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

url_map = None

The `Map` for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

```python
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def to_python(self, value):
        return value.split(',')
    def to_url(self, values):
        return ','.join(super(ListConverter, self).to_url(value)
                        for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter
```

url_rule_class

The rule object to use for URL rules created. This is used by `add_url_rule()`. Defaults to `werkzeug.routing.Rule`.

New in version 0.7.

alias of `Rule`

url_value_preprocessor(*f*)

Register a URL value preprocessor function for all view functions in the application. These functions will be called before the `before_request()` functions.

The function can modify the values captured from the matched url before they are passed to the view. For example, this can be used to pop a common language code value and place it in g rather than pass it to every view.

The function is passed the endpoint name and values dict. The return value is ignored.

url_value_preprocessors = None

A dictionary with lists of functions that are called before the `before_request_funcs` functions. The key of the dictionary is the name of the blueprint this function is active for, or `None` for all requests. To register a function, use `url_value_preprocessor()`.

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 app object. 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: Teardown events for the request and app contexts are called even if an unhandled error occurs. Other events may not be called depending on when an error occurs during dispatch. 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.

2.1.2 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, root_path=None*)

Represents a blueprint. A blueprint is an object that records functions that will be called with the BlueprintSetupState later to register functions or other things on the main application. See *Modular Applications with Blueprints* for more information.

New in version 0.7.

add_app_template_filter(*f*, *name=None*)
Register a custom template filter, available application wide. Like `Flask.add_template_filter()` but for a blueprint. Works exactly like the `app_template_filter()` decorator.

> **Parameters name** – the optional name of the filter, otherwise the function name will be used.

add_app_template_global(*f*, *name=None*)
Register a custom template global, available application wide. Like `Flask.add_template_global()` but for a blueprint. Works exactly like the `app_template_global()` decorator.

New in version 0.10.

> **Parameters name** – the optional name of the global, otherwise the function name will be used.

add_app_template_test(*f*, *name=None*)
Register a custom template test, available application wide. Like `Flask.add_template_test()` but for a blueprint. Works exactly like the `app_template_test()` decorator.

New in version 0.10.

> **Parameters name** – the optional name of the test, otherwise the function name will be used.

add_url_rule(*rule*, *endpoint=None*, *view_func=None*, ***options*)
Like `Flask.add_url_rule()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

after_app_request(*f*)
Like `Flask.after_request()` but for a blueprint. Such a function is executed after each request, even if outside of the blueprint.

after_request(*f*)
Like `Flask.after_request()` but for a blueprint. This function is only executed after each request that is handled by a function of that blueprint.

app_context_processor(*f*)
Like `Flask.context_processor()` but for a blueprint. Such a function is executed each request, even if outside of the blueprint.

app_errorhandler(*code*)
Like `Flask.errorhandler()` but for a blueprint. This handler is used for all requests, even if outside of the blueprint.

app_template_filter(*name=None*)
Register a custom template filter, available application wide. Like `Flask.template_filter()` but for a blueprint.

> **Parameters name** – the optional name of the filter, otherwise the function name will be used.

app_template_global(*name=None*)
Register a custom template global, available application wide. Like `Flask.template_global()` but for a blueprint.

New in version 0.10.

> **Parameters name** – the optional name of the global, otherwise the function name will be used.

app_template_test(*name=None*)
Register a custom template test, available application wide. Like `Flask.template_test()` but for a blueprint.

New in version 0.10.

> **Parameters name** – the optional name of the test, otherwise the function name will be used.

app_url_defaults(*f*)
Same as `url_defaults()` but application wide.

app_url_value_preprocessor(*f*)
Same as `url_value_preprocessor()` but application wide.

before_app_first_request(*f*)
Like `Flask.before_first_request()`. Such a function is executed before the first request to the application.

before_app_request(*f*)
Like `Flask.before_request()`. Such a function is executed before each request, even if outside of a blueprint.

before_request(*f*)
Like `Flask.before_request()` but for a blueprint. This function is only executed before each request that is handled by a function of that blueprint.

context_processor(*f*)
Like `Flask.context_processor()` but for a blueprint. This function is only executed for requests handled by a blueprint.

endpoint(*endpoint*)
Like `Flask.endpoint()` but for a blueprint. This does not prefix the endpoint with the blueprint name, this has to be done explicitly by the user of this method. If the endpoint is prefixed with a . it will be registered to the current blueprint, otherwise it's an application independent endpoint.

errorhandler(*code_or_exception*)
Registers an error handler that becomes active for this blueprint only. Please be aware that routing does not happen local to a blueprint so an error handler for 404 usually is not handled by a blueprint unless it is caused inside a view function. Another special case is the 500 internal server error which is always looked up from the application.

Otherwise works as the `errorhandler()` decorator of the `Flask` object.

get_send_file_max_age(*filename*)

Provides default cache_timeout for the `send_file()` functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`.

Static file functions such as `send_from_directory()` use this function, and `send_file()` calls this function on `current_app` when the given cache_timeout is `None`. If a cache_timeout is given in `send_file()`, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```python
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

New in version 0.9.

has_static_folder

This is `True` if the package bound object's container has a folder for static files.

New in version 0.5.

import_name = None

The name of the package or module that this app belongs to. Do not change this once it is set by the constructor.

jinja_loader

The Jinja loader for this package bound object.

New in version 0.5.

json_decoder = None

Blueprint local JSON decoder class to use. Set to `None` to use the app's `json_decoder`.

json_encoder = None

Blueprint local JSON decoder class to use. Set to `None` to use the app's `json_encoder`.

make_setup_state(*app*, *options*, *first_registration=False*)

Creates an instance of `BlueprintSetupState()` object that is later passed to the register callback functions. Subclasses can override this to return a subclass of the setup state.

open_resource(*resource*, *mode='rb'*)

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

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the schema.sql file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

> **Parameters**
>
> - **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
>
> - **mode** – resource file opening mode, default is 'rb'.

record(*func*)

Registers a function that is called when the blueprint is registered on the application. This function is called with the state as argument as returned by the make_setup_state() method.

record_once(*func*)

Works like record() but wraps the function in another function that will ensure the function is only called once. If the blueprint is registered a second time on the application, the function passed is not called.

register(*app*, *options*, *first_registration=False*)

Called by Flask.register_blueprint() to register all views and callbacks registered on the blueprint with the application. Creates a BlueprintSetupState and calls each record() callback with it.

> **Parameters**
>
> - **app** – The application this blueprint is being registered with.
>
> - **options** – Keyword arguments forwarded from register_blueprint().
>
> - **first_registration** – Whether this is the first time this blueprint has been registered on the application.

register_error_handler(*code_or_exception*, *f*)

Non-decorator version of the errorhandler() error attach function, akin to the register_error_handler() application-wide function of the Flask object but for error handlers limited to this blueprint.

New in version 0.11.

root_path = None
> Absolute path to the package on the filesystem. Used to look up resources contained in the package.

route(*rule, **options*)
> Like `Flask.route()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

send_static_file(*filename*)
> Function used internally to send static files from the static folder to the browser.
>
> New in version 0.5.

static_folder
> The absolute path to the configured static folder.

static_url_path
> The URL prefix that the static route will be registered for.

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.

template_folder = None
> Location of the template files to be added to the template lookup. `None` if templates should not be added.

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.

2.1.3 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.

environ
> The underlying WSGI environment.

path

full_path

script_root

url

base_url

url_root
> Provides different ways to look at the current IRI. Imagine your application is listening on the following application root:

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

> And a user requests the following URI:

```
http://www.example.com/myapplication/%CF%80/page.html?x=y
```

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

path	u'/π/page.html'
full_path	u'/π/page.html?x=y'
script_root	u'/myapplication'
base_url	u'http://www.example.com/myapplication/π/page.html'
url	u'http://www.example.com/myapplication/π/page.html?x=y'
url_root	u'http://www.example.com/myapplication/'

accept_charsets
> List of charsets this client supports as CharsetAccept object.

accept_encodings
> List of encodings this client accepts. Encodings in a HTTP term are compression encodings such as gzip. For charsets have a look at accept_charset.

accept_languages
> List of languages this client accepts as LanguageAccept object.

accept_mimetypes
> List of mimetypes this client supports as MIMEAccept object.

access_route
> If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.

application(*f*)
> Decorate a function as responder that accepts the request as first argument. This works like the responder() decorator but the function is passed the

request object as first argument and the request object will be closed automatically:

```
@Request.application
def my_wsgi_app(request):
    return Response('Hello World!')
```

As of Werkzeug 0.14 HTTP exceptions are automatically caught and converted to responses instead of failing.

> **Parameters** `f` – the WSGI callable to decorate
>
> **Returns** a new WSGI callable

args
> The parsed URL parameters (the part in the URL after the question mark).
>
> By default an `ImmutableMultiDict` is returned from this function. This can be changed by setting `parameter_storage_class` to a different type. This might be necessary if the order of the form data is important.

authorization
> The *Authorization* object in parsed form.

base_url
> Like `url` but without the querystring See also: `trusted_hosts`.

blueprint
> The name of the current blueprint

cache_control
> A `RequestCacheControl` object for the incoming cache control headers.

close()
> Closes associated resources of this request object. This closes all file handles explicitly. You can also use the request object in a with statement which will automatically close it.
>
> New in version 0.9.

content_encoding
> The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the Content-Type header field.
>
> New in version 0.9.

content_length
> The Content-Length entity-header field indicates the size of the entity-body in bytes or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.

content_md5

The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body. (Note: a MIC is good for detecting accidental modification of the entity-body in transit, but is not proof against malicious attacks.)

New in version 0.9.

content_type

The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

cookies

A `dict` with the contents of all cookies transmitted with the request.

data

Contains the incoming request data as string in case it came with a mime-type Werkzeug does not handle.

date

The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.

dict_storage_class

alias of `ImmutableTypeConversionDict`

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`.

files

`MultiDict` object containing all uploaded files. Each key in `files` is the name from the `<input type="file" name="">`. Each value in `files` is a Werkzeug `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.

Note that `files` will only contain data if the request method was POST, PUT or PATCH and the `<form>` that posted to the request had `enctype="multipart/form-data"`. It will be empty otherwise.

See the `MultiDict` / `FileStorage` documentation for more details about the used data structure.

form

The form parameters. By default an `ImmutableMultiDict` is returned from this function. This can be changed by setting `parameter_storage_class` to a different type. This might be necessary if the order of the form data is important.

Please keep in mind that file uploads will not end up here, but instead in the `files` attribute.

Changed in version 0.9: Previous to Werkzeug 0.9 this would only contain form data for POST and PUT requests.

form_data_parser_class
alias of FormDataParser

from_values(**args*, ***kwargs*)
Create a new request object based on the values provided. If environ is given missing values are filled from there. This method is useful for small scripts when you need to simulate a request from an URL. Do not use this method for unittesting, there is a full featured client object (Client) that allows to create multipart requests, support for cookies etc.

This accepts the same options as the EnvironBuilder.

Changed in version 0.5: This method now accepts the same arguments as EnvironBuilder. Because of this the *environ* parameter is now called *environ_overrides*.

> **Returns** request object

full_path
Requested path as unicode, including the query string.

get_data(*cache=True*, *as_text=False*, *parse_form_data=False*)
This reads the buffered incoming data from the client into one bytestring. By default this is cached but that behavior can be changed by setting *cache* to *False*.

Usually it's a bad idea to call this method without checking the content length first as a client could send dozens of megabytes or more to cause memory problems on the server.

Note that if the form data was already parsed this method will not return anything as form data parsing does not cache the data like this method does. To implicitly invoke form data parsing function set *parse_form_data* to *True*. When this is done the return value of this method will be an empty string if the form parser handles the data. This generally is not necessary as if the whole data is cached (which is the default) the form parser will used the cached data to parse the form data. Please be generally aware of checking the content length first in any case before calling this method to avoid exhausting server memory.

If *as_text* is set to *True* the return value will be a decoded unicode string.

New in version 0.9.

get_json(*force=False*, *silent=False*, *cache=True*)
Parse and return the data as JSON. If the mimetype does not indicate JSON (application/json, see is_json()), this returns None unless force is true.

If parsing fails, `on_json_loading_failed()` is called and its return value is used as the return value.

> **Parameters**
>
> - **force** – Ignore the mimetype and always try to parse JSON.
> - **silent** – Silence parsing errors and return `None` instead.
> - **cache** – Store the parsed JSON to return for subsequent calls.

headers
> The headers from the WSGI environ as immutable `EnvironHeaders`.

host
> Just the host including the port if available. See also: `trusted_hosts`.

host_url
> Just the host with scheme as IRI. See also: `trusted_hosts`.

if_match
> An object containing all the etags in the *If-Match* header.
>
> > **Return type** `ETags`

if_modified_since
> The parsed *If-Modified-Since* header as datetime object.

if_none_match
> An object containing all the etags in the *If-None-Match* header.
>
> > **Return type** `ETags`

if_range
> The parsed *If-Range* header.
>
> New in version 0.7.
>
> > **Return type** `IfRange`

if_unmodified_since
> The parsed *If-Unmodified-Since* header as datetime object.

is_json
> Check if the mimetype indicates JSON data, either `application/json` or `application/*+json`.
>
> New in version 0.11.

is_multiprocess
> boolean that is *True* if the application is served by a WSGI server that spawns multiple processes.

is_multithread
> boolean that is *True* if the application is served by a multithreaded WSGI server.

is_run_once
> boolean that is *True* if the application will be executed only once in a process lifetime. This is the case for CGI for example, but it's not guaranteed that the execution only happens one time.

is_secure
> *True* if the request is secure.

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.
>
> Deprecated since version 0.13: `X-Requested-With` is not standard and is unreliable.

json
> This will contain the parsed JSON data if the mimetype indicates JSON (`application/json`, see `is_json()`), otherwise it will be `None`.

list_storage_class
> alias of `ImmutableList`

make_form_data_parser()
> Creates the form data parser. Instantiates the `form_data_parser_class` with some parameters.
>
> New in version 0.8.

max_content_length
> Read-only view of the `MAX_CONTENT_LENGTH` config key.

max_forwards
> The Max-Forwards request-header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server.

method
> The request method. (For example `'GET'` or `'POST'`).

mimetype
> Like `content_type`, but without parameters (eg, without charset, type etc.) and always lowercase. For example if the content type is `text/HTML; charset=utf-8` the mimetype would be `'text/html'`.

mimetype_params
> The mimetype parameters as dict. For example if the content type is `text/html; charset=utf-8` the params would be `{'charset': 'utf-8'}`.

on_json_loading_failed(*e*)
> Called if `get_json()` parsing fails and isn't silenced. If this method returns a value, it is used as the return value for `get_json()`. The default implementation raises a `BadRequest` exception.

Changed in version 0.10: Raise a `BadRequest` error instead of returning an error message as JSON. If you want that behavior you can add it by subclassing.

New in version 0.8.

parameter_storage_class
   alias of `ImmutableMultiDict`

path
   Requested path as unicode. This works a bit like the regular path info in the WSGI environment but will always include a leading slash, even if the URL root is accessed.

pragma
   The Pragma general-header field is used to include implementation-specific directives that might apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.

query_string
   The URL parameters as raw bytestring.

range
   The parsed *Range* header.

   New in version 0.7.

   > **Return type** Range

referrer
   The Referer[sic] request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled).

remote_addr
   The remote address of the client.

remote_user
   If the server supports user authentication, and the script is protected, this attribute contains the username the user has authenticated as.

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.

scheme
   URL scheme (http or https).

   New in version 0.7.

script_root
   The root path of the script without the trailing slash.

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.

Unlike `input_stream` this stream is properly guarded that you can't accidentally read past the length of the input. Werkzeug will internally always refer to this stream to read data which makes it possible to wrap this object with a stream that does filtering.

Changed in version 0.9: This stream is now always available but might be consumed by the form parser later on. Previously the stream was only set if no parsing happened.

url

The reconstructed current URL as IRI. See also: `trusted_hosts`.

url_charset

The charset that is assumed for URLs. Defaults to the value of `charset`.

New in version 0.6.

url_root

The full URL root (with hostname), this is the application root as IRI. See also: `trusted_hosts`.

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. Though if the request's method was invalid for the URL rule, the valid list is available in `routing_exception.valid_methods` instead (an attribute of the Werkzeug exception `MethodNotAllowed`) because the request was never internally bound.

New in version 0.6.

user_agent

The current user agent.

values

A `werkzeug.datastructures.CombinedMultiDict` that combines args and form.

view_args = None

A dict of view arguments that matched the request. If an exception happened when matching, this will be `None`.

want_form_data_parsed

Returns True if the request method carries content. As of Werkzeug 0.9 this will be the case if a content type is transmitted.

New in version 0.8.

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.

2.1.4 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.
>
>   Changed in version 1.0: JSON support is added to the response, like the request. This is useful when testing to get the test client response data as JSON.
>
>   Changed in version 1.0: Added max_cookie_size.
>
>   **headers**
>   >   A Headers object representing the response headers.
>
>   **status**
>   >   A string with a response status.
>
>   **status_code**
>   >   The response status as integer.
>
>   **data**
>   >   A descriptor that calls get_data() and set_data(). This should not be used and will eventually get deprecated.
>
>   **get_json**(*force=False, silent=False, cache=True*)
>   >   Parse and return the data as JSON. If the mimetype does not indicate JSON (application/json, see is_json()), this returns None unless force is true. If parsing fails, on_json_loading_failed() is called and its return value is used as the return value.
>   >
>   >   >   **Parameters**
>   >   >
>   >   >   - **force** – Ignore the mimetype and always try to parse JSON.
>   >   >
>   >   >   - **silent** – Silence parsing errors and return None instead.

- **cache** – Store the parsed JSON to return for subsequent calls.

is_json

Check if the mimetype indicates JSON data, either `application/json` or `application/*+json`.

New in version 0.11.

max_cookie_size

Read-only view of the `MAX_COOKIE_SIZE` config key.

See `max_cookie_size` in Werkzeug's docs.

mimetype

The mimetype (content type without charset etc.)

set_cookie(*key, value='', max_age=None, expires=None, path='/', domain=None, secure=False, httponly=False, samesite=None*)

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.

A warning is raised if the size of the cookie header exceeds `max_cookie_size`, but the header will still be set.

Parameters

- **key** – the key (name) of the cookie to be set.

- **value** – the value of the cookie.

- **max_age** – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client's browser session.

- **expires** – should be a *datetime* object or UNIX timestamp.

- **path** – limits the cookie to a given path, per default it will span the whole domain.

- **domain** – if you want to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domain `www.example.com`, `foo.example.com` etc. Otherwise, a cookie will only be readable by the domain that set it.

- **secure** – If *True*, the cookie will only be available via HTTPS

- **httponly** – disallow JavaScript to access the cookie. This is an extension to the cookie standard and probably not supported by all browsers.

- **samesite** – Limits the scope of the cookie such that it will only be attached to requests if those requests are "same-site".

2.1.5 Sessions

If you have set `Flask.secret_key` (or configured it from `SECRET_KEY`) you can use sessions in Flask applications. A session makes it possible to remember information from one request to another. The way Flask does this is by using a signed cookie. The user can look at the session contents, but can't 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.

2.1.6 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*)
> Returns the domain that should be set for the session cookie.

> Uses SESSION_COOKIE_DOMAIN if it is configured, otherwise falls back to detecting the domain based on SERVER_NAME.

> Once detected (or if not set at all), SESSION_COOKIE_DOMAIN is updated to avoid re-running the logic.

get_cookie_httponly(*app*)
> Returns True if the session cookie should be httponly. This currently just returns the value of the SESSION_COOKIE_HTTPONLY config var.

get_cookie_path(*app*)
> Returns the path for which the cookie should be valid. The default implementation uses the value from the SESSION_COOKIE_PATH config var if it's set, and falls back to APPLICATION_ROOT or uses / if it's None.

get_cookie_samesite(*app*)
> Return 'Strict' or 'Lax' if the cookie should use the SameSite attribute. This currently just returns the value of the SESSION_COOKIE_SAMESITE setting.

get_cookie_secure(*app*)
> Returns True if the cookie should be secure. This currently just returns the value of the SESSION_COOKIE_SECURE setting.

get_expiration_time(*app, session*)
> A helper method that returns an expiration date for the session or None if the session is linked to the browser session. The default implementation returns now + the permanent session lifetime configured on the application.

is_null_session(*obj*)
> Checks if a given object is a null session. Null sessions are not asked to be saved.

This checks if the object is an instance of `null_session_class` by default.

make_null_session(*app*)

Creates a null session which acts as a replacement object if the real session support could not be loaded due to a configuration error. This mainly aids the user experience because the job of the null session is to still support lookup without complaining but modifications are answered with a helpful error message of what failed.

This creates an instance of `null_session_class` by default.

null_session_class

make_null_session() will look here for the class that should be created when a null session is requested. Likewise the is_null_session() method will perform a typecheck against this type.

alias of `NullSession`

open_session(*app*, *request*)

This method has to be implemented and must either return `None` in case the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on `SessionMixin`.

pickle_based = False

A flag that indicates if the session interface is pickle based. This can be used by Flask extensions to make a decision in regards to how to deal with the session object.

New in version 0.10.

save_session(*app*, *session*, *response*)

This is called for actual sessions returned by open_session() at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.

should_set_cookie(*app*, *session*)

Used by session backends to determine if a `Set-Cookie` header should be set for this session cookie for this response. If the session has been modified, the cookie is set. If the session is permanent and the `SESSION_REFRESH_EACH_REQUEST` config is true, the cookie is always set.

This check is usually skipped if the session was deleted.

New in version 0.11.

class flask.sessions.**SecureCookieSessionInterface**

The default session interface that stores sessions in signed cookies through the `itsdangerous` module.

static digest_method()

the hash function to use for the signature. The default is sha1

key_derivation = 'hmac'

the name of the itsdangerous supported key derivation. The default is

hmac.

salt = 'cookie-session'
the salt that should be applied on top of the secret key for the signing of cookie based sessions.

serializer = <flask.json.tag.TaggedJSONSerializer object>
A python serializer for the payload. The default is a compact JSON derived serializer with support for some extra Python types such as datetime objects or tuples.

session_class
alias of `SecureCookieSession`

class `flask.sessions.`**SecureCookieSession**(*initial=None*)
Base class for sessions based on signed cookies.

This session backend will set the `modified` and `accessed` attributes. It cannot reliably track whether a session is new (vs. empty), so `new` remains hard coded to `False`.

accessed = False
When data is read or written, this is set to `True`. Used by header, which allows caching proxies to cache different pages for different users.

modified = False
When data is changed, this is set to `True`. Only the session dictionary itself is tracked; if the session contains mutable data (for example a nested dict) then this must be set to `True` manually when modifying that data. The session cookie will only be written to the response if this is `True`.

class `flask.sessions.`**NullSession**(*initial=None*)
Class used to generate nicer error messages if sessions are not available. Will still allow read-only access to the empty session but fail on setting.

class `flask.sessions.`**SessionMixin**
Expands a basic dictionary with session attributes.

accessed = True
Some implementations can detect when session data is read or written and set this when that happens. The mixin default is hard coded to `True`.

modified = True
Some implementations can detect changes to the session and set this when that happens. The mixin default is hard coded to `True`.

new = False
Some implementations can detect whether a session is newly created, but that is not guaranteed. Use with caution. The mixin

permanent
This reflects the '`_permanent`' key in the dict.

Notice

The PERMANENT_SESSION_LIFETIME config key can also be an integer starting with Flask 0.8. Either catch this down yourself or use the permanent_session_lifetime attribute on the app which converts the result to an integer automatically.

2.1.7 Test Client

class flask.testing.**FlaskClient**(*args, **kwargs*)

Works like a regular Werkzeug test client but has some knowledge about how Flask works to defer the cleanup of the request context stack to the end of a with body when used in a with statement. For general information about how to use this class refer to werkzeug.test.Client.

Changed in version 0.12: *app.test_client()* includes preset default environment, which can be set after instantiation of the *app.test_client()* object in *client.environ_base*.

Basic usage is outlined in the *Testing Flask Applications* chapter.

session_transaction(*args, **kwargs*)

When used in combination with a with statement this opens a session transaction. This can be used to modify the session that the test client uses. Once the with block is left the session is stored back.

```
with client.session_transaction() as session:
    session['value'] = 42
```

Internally this is implemented by going through a temporary test request context and since session handling could depend on request variables this function accepts the same arguments as test_request_context() which are directly passed through.

2.1.8 Test CLI Runner

class flask.testing.**FlaskCliRunner**(*app, **kwargs*)

A CliRunner for testing a Flask app's CLI commands. Typically created using test_cli_runner(). See *Testing CLI Commands*.

invoke(*cli=None, args=None, **kwargs*)

Invokes a CLI command in an isolated environment. See CliRunner.invoke for full method documentation. See *Testing CLI Commands* for examples.

If the obj argument is not given, passes an instance of ScriptInfo that knows how to load the Flask app being tested.

Parameters

- **cli** – Command object to invoke. Default is the app's cli group.

- **args** – List of strings to invoke the command with.

Returns a `Result` object.

2.1.9 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`**

A namespace object that can store data during an *application context*. This is an instance of `Flask.app_ctx_globals_class`, which defaults to `ctx._AppCtxGlobals`.

This is a good place to store resources during a request. During testing, you can use the *Faking Resources and Context* pattern to pre-configure such resources.

This is a proxy. See *Notes On Proxies* for more information.

Changed in version 0.10: Bound to the application context instead of the request context.

`class` `flask.ctx.`**`_AppCtxGlobals`**

A plain object. Used as a namespace for storing data during an application context.

Creating an app context automatically creates this object, which is made available as the g proxy.

`'key' in g`

Check whether an attribute is present.

New in version 0.10.

`iter(g)`

Return an iterator over the attribute names.

New in version 0.10.

`get`(*name*, *default=None*)

Get an attribute by name, or a default value. Like `dict.get()`.

Parameters

- **`name`** – Name of attribute to get.

- **`default`** – Value to return if the attribute is not present.

New in version 0.10.

`pop`(*name*, *default=<object object>*)

Get and remove an attribute by name. Like `dict.pop()`.

Parameters

- **`name`** – Name of attribute to pop.

> • **default** – Value to return if the attribute is not present, instead of raise a `KeyError`.

New in version 0.11.

setdefault(*name*, *default=None*)
> Get the value of an attribute if it is present, otherwise set and return a default value. Like `dict.setdefault()`.

> > **Parameters name** – Name of attribute to get.

> > **Param** default: Value to set and return if the attribute is not present.

New in version 0.11.

2.1.10 Useful Functions and Classes

flask.**current_app**
> A proxy to the application handling the current request. This is useful to access the application without needing to import it, or if it can't be imported, such as when using the application factory pattern or in blueprints and extensions.

> This is only available when an *application context* is pushed. This happens automatically during requests and CLI commands. It can be controlled manually with `app_context()`.

> This is a proxy. See *Notes On Proxies* for more information.

flask.**has_request_context**()
> If you have code that wants to test if a request context is there or not this function can be used. For instance, you may want to take advantage of request information if the request object is available, but fail silently if it is unavailable.

```python
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):

```python
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and request:
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

New in version 0.7.

flask.**copy_current_request_context**(*f*)

> A helper function that decorates a function to retain the current request context. This is useful when working with greenlets. The moment the function is decorated a copy of the request context is created and then pushed when the function is called.
>
> Example:

```
import gevent
from flask import copy_current_request_context

@app.route('/')
def index():
    @copy_current_request_context
    def do_some_work():
        # do some work here, it can access flask.request like you
        # would otherwise in the view function.
        ...
    gevent.spawn(do_some_work)
    return 'Regular response'
```

> New in version 0.10.

flask.**has_app_context**()

> Works like has_request_context() but for the application context. You can also just do a boolean check on the current_app object instead.
>
> New in version 0.9.

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

> Generates a URL to the given endpoint with the method provided.
>
> Variable arguments that are unknown to the target endpoint are appended to the generated URL as query arguments. If the value of a query argument is None, the whole pair is skipped. In case blueprints are active you can shortcut references to the same blueprint by prefixing the local endpoint with a dot (.).
>
> This will reference the index function local to the current blueprint:

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

> For more information, head over to the *Quickstart*.
>
> To integrate applications, Flask has a hook to intercept URL build errors through Flask.url_build_error_handlers. The *url_for* function results in a BuildError when the current app does not have a URL for the given endpoint and values. When it does, the current_app calls its url_build_error_handlers if it is not None, which can return a string to use as the result of *url_for* (instead of *url_for*'s default to raise the BuildError exception) or re-raise the exception. An example:

```
def external_url_handler(error, endpoint, values):
    "Looks up an external URL when `url_for` cannot build a URL."
    # This is an example of hooking the build_error_handler.
    # Here, lookup_url is some utility function you've built
    # which looks up the endpoint in some external URL registry.
    url = lookup_url(endpoint, **values)
    if url is None:
        # External lookup did not have a URL.
        # Re-raise the BuildError, in context of original traceback.
        exc_type, exc_value, tb = sys.exc_info()
        if exc_value is error:
            raise exc_type, exc_value, tb
        else:
            raise error
    # url_for will use this result, instead of raising BuildError.
    return url

app.url_build_error_handlers.append(external_url_handler)
```

Here, *error* is the instance of BuildError, and *endpoint* and *values* are the arguments passed into *url_for*. Note that this is for building URLs outside the current application, and not for handling 404 NotFound errors.

New in version 0.10: The *_scheme* parameter was added.

New in version 0.9: The *_anchor* and *_method* parameters were added.

New in version 0.9: Calls Flask.handle_build_error() on BuildError.

> **Parameters**
>
> - **endpoint** – the endpoint of the URL (name of the function)
>
> - **values** – the variable arguments of the URL rule
>
> - **_external** – if set to True, an absolute URL is generated. Server address can be changed via SERVER_NAME configuration variable which defaults to *localhost*.
>
> - **_scheme** – a string specifying the desired URL scheme. The *_external* parameter must be set to True or a ValueError is raised. The default behavior uses the same scheme as the current request, or PREFERRED_URL_SCHEME from the *app configuration* if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.
>
> - **_anchor** – if provided this is added as anchor to the URL.
>
> - **_method** – if provided this explicitly specifies an HTTP method.

flask.**abort**(*status*, **args*, ***kwargs*)

> Raises an HTTPException for the given status code or WSGI application:

```
abort(404)  # 404 Not Found
abort(Response('Hello World'))
```

Can be passed a WSGI application or a status code. If a status code is given it's
looked up in the list of exceptions and will raise that exception, if passed a WSGI
application it will wrap it in a proxy WSGI exception and raise that:

```
abort(404)
abort(Response('Hello World'))
```

flask.**redirect**(*location*, *code=302*, *Response=None*)

Returns a response object (a WSGI application) that, if called, redirects the client
to the target location. Supported codes are 301, 302, 303, 305, and 307. 300 is not
supported because it's not a real redirect and 304 because it's the answer for a
request with a request with defined If-Modified-Since headers.

New in version 0.6: The location can now be a unicode string that is encoded
using the iri_to_uri() function.

New in version 0.10: The class used for the Response object can now be passed
in.

Parameters

- **location** – the location the response should redirect to.

- **code** – the redirect status code. defaults to 302.

- **Response** (class) – a Response class to use when instantiating
  a response. The default is werkzeug.wrappers.Response if un-
  specified.

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 func-
tion. This for example creates a response with a 404 error code:

```
response = make_response(render_template('not_found.html'), 404)
```

The other use case of this function is to force the return value of a view function into a response which is helpful with view decorators:

```
response = make_response(view_function())
response.headers['X-Parachutes'] = 'parachutes are cool'
```

Internally this function does the following things:

- if no arguments are passed, it creates a new response argument

- if one argument is passed, `flask.Flask.make_response()` is invoked with it.

- if more than one argument is passed, the arguments are passed to the `flask.Flask.make_response()` function as tuple.

New in version 0.6.

flask.**after_this_request**(*f*)

Executes a function after this request. This is useful to modify response objects. The function is passed the response object and has to return the same or a new one.

Example:

```
@app.route('/')
def index():
    @after_this_request
    def add_header(response):
        response.headers['X-Foo'] = 'Parachute'
        return response
    return 'Hello World!'
```

This is more useful if a function other than the view function wants to modify a response. For instance think of a decorator that wants to add some headers without converting the return value into a response object.

New in version 0.9.

flask.**send_file**(*filename_or_fp*, *mimetype=None*, *as_attachment=False*, *attachment_filename=None*, *add_etags=True*, *cache_timeout=None*, *conditional=False*, *last_modified=None*)

Sends the contents of a file to the client. This will use the most efficient method available and configured. By default it will try to use the WSGI server's file_wrapper support. Alternatively you can set the application's `use_x_sendfile` attribute to `True` to directly emit an X-Sendfile header. This however requires support of the underlying webserver for X-Sendfile.

By default it will try to guess the mimetype for you, but you can also explicitly provide one. For extra security you probably want to send certain files as attachment (HTML for instance). The mimetype guessing requires a *filename* or an

attachment_filename to be provided.

ETags will also be attached automatically if a *filename* is provided. You can turn this off by setting *add_etags=False*.

If *conditional=True* and *filename* is provided, this method will try to upgrade the response stream to support range requests. This will allow the request to be answered with partial content response.

Please never pass filenames to this function from user sources; you should use send_from_directory() instead.

New in version 0.2.

New in version 0.5: The *add_etags*, *cache_timeout* and *conditional* parameters were added. The default behavior is now to attach etags.

Changed in version 0.7: mimetype guessing and etag support for file objects was deprecated because it was unreliable. Pass a filename if you are able to, otherwise attach an etag yourself. This functionality will be removed in Flask 1.0

Changed in version 0.9: cache_timeout pulls its default from application config, when None.

Changed in version 0.12: The filename is no longer automatically inferred from file objects. If you want to use automatic mimetype and etag support, pass a filepath via *filename_or_fp* or *attachment_filename*.

Changed in version 0.12: The *attachment_filename* is preferred over *filename* for MIME-type detection.

Changed in version 0.13: UTF-8 filenames, as specified in RFC 2231, are supported.

> **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. If a file path is given, auto detection happens as fallback, otherwise an error will be raised.
> >
> > • `as_attachment` – set to `True` if you want to send this file with a `Content-Disposition: attachment` header.
> >
> > • `attachment_filename` – the filename for the attachment if it differs from the file's filename.
> >
> > • `add_etags` – set to `False` to disable attaching of etags.
> >
> > • `conditional` – set to `True` to enable conditional responses.

- **cache_timeout** – the timeout in seconds for the headers. When None (default), this value is set by `get_send_file_max_age()` of `current_app`.

- **last_modified** – set the `Last-Modified` header to this value, a `datetime` or timestamp. If a file was passed, this overrides its mtime.

`flask.`**`send_from_directory`**(*directory*, *filename*, ***options*)

Send a file from a given directory with `send_file()`. This is a secure way to quickly expose static files from an upload folder or something similar.

Example usage:

```python
@app.route('/uploads/<path:filename>')
def download_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename, as_attachment=True)
```

Sending files and Performance

It is strongly recommended to activate either `X-Sendfile` support in your webserver or (if no authentication happens) to tell the webserver to serve files for the given path on its own without calling into the web application for improved performance.

New in version 0.5.

> **Parameters**
>
> - **directory** – the directory where all the files are stored.
>
> - **filename** – the filename relative to that directory to download.
>
> - **options** – optional keyword arguments that are directly forwarded to `send_file()`.

`flask.`**`safe_join`**(*directory*, **pathnames*)

Safely join *directory* and zero or more untrusted *pathnames* components.

Example usage:

```python
@app.route('/wiki/<path:filename>')
def wiki_page(filename):
    filename = safe_join(app.config['WIKI_FOLDER'], filename)
    with open(filename, 'rb') as fd:
        content = fd.read()  # Read and process the file content...
```

> **Parameters**
>
> - **directory** – the trusted base directory.
>
> - **pathnames** – the untrusted pathnames relative to that directory.

Raises NotFound if one or more passed paths fall out of its boundaries.

flask.**escape**(*s*) → markup
> Convert the characters &, <, >, ', and " in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

class flask.**Markup**
> Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the *__html__* interface a couple of frameworks and web applications use. Markup is a direct subclass of *unicode* and provides all the methods of *unicode* just that it escapes arguments passed and always returns *Markup*.
>
> The *escape* function returns markup objects so that double escaping can't happen.
>
> The constructor of the Markup class can be used for three different things: When passed an unicode object it's assumed to be safe, when passed an object with an HTML representation (has an *__html__* method) that representation is used, otherwise the object passed is converted into a unicode string and then assumed to be safe:

```
>>> Markup("Hello <em>World</em>!")
Markup(u'Hello <em>World</em>!')
>>> class Foo(object):
...   def __html__(self):
...     return '<a href="#">foo</a>'
...
>>> Markup(Foo())
Markup(u'<a href="#">foo</a>')
```

> If you want object passed being always treated as unsafe you can use the escape() classmethod to create a Markup object:

```
>>> Markup.escape("Hello <em>World</em>!")
Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')
```

> Operations on a markup string are markup aware which means that all arguments are passed through the escape() function:

```
>>> em = Markup("<em>%s</em>")
>>> em % "foo & bar"
Markup(u'<em>foo &amp; bar</em>')
>>> strong = Markup("<strong>%(text)s</strong>")
>>> strong % {'text': '<blink>hacker here</blink>'}
Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup(u'<em>Hello</em> &lt;foo&gt;')
```

> **classmethod escape**(*s*)
> > Escape the string. Works like escape() with the difference that for subclasses of Markup this function would return the correct subclass.

striptags()

> Unescape markup into an text_type string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
u'Main \xbb About'
```

unescape()

> Unescape markup again into an text_type string. This also resolves known HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

2.1.11 Message Flashing

flask.**flash**(*message*, *category='message'*)

> Flashes a message to the next request. In order to remove the flashed message from the session and to display it to the user, the template has to call get_flashed_messages().

> Changed in version 0.3: *category* parameter added.

> **Parameters**

> > - **message** – the message to be flashed.
> >
> > - **category** – the category for the message. The following values are recommended: 'message' for any kind of message, 'error' for errors, 'info' for information messages and 'warning' for warnings. However any kind of string can be used as category.

flask.**get_flashed_messages**(*with_categories=False*, *category_filter=[]*)

> Pulls all flashed messages from the session and returns them. Further calls in the same request to the function will return the same messages. By default just the messages are returned, but when *with_categories* is set to True, the return value will be a list of tuples in the form (category, message) instead.

> Filter the flashed messages to one or more categories by providing those categories in *category_filter*. This allows rendering categories in separate html blocks. The *with_categories* and *category_filter* arguments are distinct:

> - *with_categories* controls whether categories are returned with message text (True gives a tuple, where False gives just the message text).
>
> - *category_filter* filters the messages down to only those matching the provided categories.

> See *Message Flashing* for examples.

> Changed in version 0.3: *with_categories* parameter added.

Changed in version 0.9: *category_filter* parameter added.

> **Parameters**
>
> - **with_categories** – set to `True` to also receive categories.
> - **category_filter** – whitelist of categories to limit return values

2.1.12 JSON Support

Flask uses `simplejson` for the JSON implementation. Since simplejson is provided by both the standard library as well as extension, Flask will try simplejson first and then fall back to the stdlib json module. On top of that it will delegate access to the current application's JSON encoders and decoders for easier customization.

So for starters instead of doing:

```python
try:
    import simplejson as json
except ImportError:
    import json
```

You can instead just do this:

```python
from flask import json
```

For usage examples, read the `json` documentation in the standard library. The following extensions are by default applied to the stdlib's JSON module:

1. `datetime` objects are serialized as **RFC 822** strings.

2. Any object with an `__html__` method (like `Markup`) will have that method called and then the return value is serialized as string.

The `htmlsafe_dumps()` function of this json module is also available as filter called `|tojson` in Jinja2. Note that inside `script` tags no escaping must take place, so make sure to disable escaping with `|safe` if you intend to use it inside `script` tags unless you are using Flask 0.10 which implies that:

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

Auto-Sort JSON Keys

The configuration variable `JSON_SORT_KEYS` (*Configuration Handling*) can be set to false to stop Flask from auto-sorting keys. By default sorting is enabled and outside of the app context sorting is turned on.

Notice that disabling key sorting can cause issues when using content based HTTP caches and Python's hash randomization feature.

---

flask.json.**jsonify**(*args, **kwargs*)

> This function wraps dumps() to add a few enhancements that make life easier. It turns the JSON output into a Response object with the application/json mime-type. For convenience, it also converts multiple arguments into an array or multiple keyword arguments into a dict. This means that both jsonify(1,2,3) and jsonify([1,2,3]) serialize to [1,2,3].
>
> For clarity, the JSON serialization behavior has the following differences from dumps():
>
> 1. Single argument: Passed straight through to dumps().
>
> 2. Multiple arguments: Converted to an array before being passed to dumps().
>
> 3. Multiple keyword arguments: Converted to a dict before being passed to dumps().
>
> 4. Both args and kwargs: Behavior undefined and will throw an exception.
>
> Example usage:

```
from flask import jsonify

@app.route('/_get_current_user')
def get_current_user():
    return jsonify(username=g.user.username,
                   email=g.user.email,
                   id=g.user.id)
```

> This will send a JSON response like this to the browser:

```
{
    "username": "admin",
    "email": "admin@localhost",
    "id": 42
}
```

> Changed in version 0.11: Added support for serializing top-level arrays. This introduces a security risk in ancient browsers. See *JSON Security* for details.
>
> This function's response will be pretty printed if the JSONIFY_PRETTYPRINT_REGULAR config parameter is set to True or the Flask app is running in debug mode. Compressed (not pretty) formatting currently means no indents and no spaces after separators.
>
> New in version 0.2.

flask.json.**dumps**(*obj, **kwargs*)

> Serialize obj to a JSON formatted str by using the application's configured encoder (json_encoder) if there is an application on the stack.
>
> This function can return unicode strings or ascii-only bytestrings by default which coerce into unicode strings automatically. That behavior by default is

controlled by the `JSON_AS_ASCII` configuration variable and can be overridden by the simplejson `ensure_ascii` parameter.

flask.json.**dump**(*obj*, *fp*, ***kwargs*)
    Like dumps() but writes into a file object.

flask.json.**loads**(*s*, ***kwargs*)
    Unserialize a JSON object from a string s by using the application's configured decoder (`json_decoder`) if there is an application on the stack.

flask.json.**load**(*fp*, ***kwargs*)
    Like loads() but reads from a file object.

class flask.json.**JSONEncoder**(*skipkeys=False,* *ensure_ascii=True,* *check_circular=True,* *allow_nan=True,* *sort_keys=False, indent=None, separators=None,* *encoding='utf-8', default=None*)
    The default Flask JSON encoder. This one extends the default simplejson encoder by also supporting `datetime` objects, `UUID` as well as `Markup` objects which are serialized as RFC 822 datetime strings (same as the HTTP date format). In order to support more data types override the `default()` method.

    default(*o*)
        Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a `TypeError`).

        For example, to support arbitrary iterators, you could implement default like this:

```python
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    return JSONEncoder.default(self, o)
```

class flask.json.**JSONDecoder**(*encoding=None,* *object_hook=None,* *parse_float=None,* *parse_int=None,* *parse_constant=None, strict=True, object_pairs_hook=None*)
    The default JSON decoder. This one does not change the behavior from the default simplejson decoder. Consult the `json` documentation for more information. This decoder is not only used for the load functions of this module but also `Request`.

Tagged JSON

A compact representation for lossless serialization of non-standard JSON types. `SecureCookieSessionInterface` uses this to serialize the session data, but it may be

useful in other places. It can be extended to support other types.

class flask.json.tag.**TaggedJSONSerializer**

Serializer that uses a tag system to compactly represent objects that are not JSON types. Passed as the intermediate serializer to itsdangerous.Serializer.

The following extra types are supported:

- dict
- tuple
- bytes
- Markup
- UUID
- datetime

default_tags = [<class 'flask.json.tag.TagDict'>, <class 'flask.json.tag.PassDict'>

Tag classes to bind when creating the serializer. Other tags can be added later using register().

dumps(*value*)

Tag the value and dump it to a compact JSON string.

loads(*value*)

Load data from a JSON string and deserialized any tagged objects.

register(*tag_class, force=False, index=-1*)

Register a new tag with this serializer.

> **Parameters**
>
> - **tag_class** – tag class to register. Will be instantiated with this serializer instance.
> - **force** – overwrite an existing tag. If false (default), a KeyError is raised.
> - **index** – index to insert the new tag in the tag order. Useful when the new tag is a special case of an existing tag. If -1 (default), the tag is appended to the end of the order.
>
> **Raises** KeyError – if the tag key is already registered and force is not true.

tag(*value*)

Convert a value to a tagged representation if necessary.

untag(*value*)

Convert a tagged representation back to the original type.

class flask.json.tag.**JSONTag**(*serializer*)

Base class for defining type tags for TaggedJSONSerializer.

check(*value*)
> Check if the given value should be tagged by this tag.

key = None
> The tag to mark the serialized object with. If None, this tag is only used as an intermediate step during tagging.

tag(*value*)
> Convert the value to a valid JSON type and add the tag structure around it.

to_json(*value*)
> Convert the Python object to an object that is a valid JSON type. The tag will be added later.

to_python(*value*)
> Convert the JSON representation back to the correct type. The tag will already be removed.

Let's seen an example that adds support for OrderedDict. Dicts don't have an order in Python or JSON, so to handle this we will dump the items as a list of [key, value] pairs. Subclass JSONTag and give it the new key ' od' to identify the type. The session serializer processes dicts first, so insert the new tag at the front of the order since OrderedDict must be processed before dict.

```python
from flask.json.tag import JSONTag

class TagOrderedDict(JSONTag):
    __slots__ = ('serializer',)
    key = ' od'

    def check(self, value):
        return isinstance(value, OrderedDict)

    def to_json(self, value):
        return [[k, self.serializer.tag(v)] for k, v in iteritems(value)]

    def to_python(self, value):
        return OrderedDict(value)

app.session_interface.serializer.register(TagOrderedDict, 0)
```

2.1.13 Template Rendering

flask.**render_template**(*template_name_or_list*, ***context*)
> Renders a template from the template folder with the given context.
>
> > **Parameters**
> >
> > - **template_name_or_list** – the name of the template to be rendered, or an iterable with template names the first one existing will be rendered

- **context** – the variables that should be available in the context of the template.

flask.**render_template_string**(*source, **context*)

Renders a template from the given template source string with the given context. Template variables will be autoescaped.

Parameters

- **source** – the source code of the template to be rendered
- **context** – the variables that should be available in the context of the template.

flask.**get_template_attribute**(*template_name, attribute*)

Loads a macro (or variable) a template exports. This can be used to invoke a macro from within Python code. If you for example have a template named _cider.html with the following contents:

```
{% macro hello(name) %}Hello {{ name }}!{% endmacro %}
```

You can access this from Python code like this:

```
hello = get_template_attribute('_cider.html', 'hello')
return hello('World')
```

New in version 0.2.

Parameters

- **template_name** – the name of the template
- **attribute** – the name of the variable of macro to access

2.1.14 Configuration

class flask.**Config**(*root_path, defaults=None*)

Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

Either you can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Or alternatively you can define the configuration options in the module that calls from_object() or provide an import path to a module that should be loaded. It is also possible to tell it to use the same module and with that provide the configuration values just before the call:

```
DEBUG = True
SECRET_KEY = 'development key'
app.config.from_object(__name__)
```

In both cases (loading from any Python file or loading from modules), only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

```
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS X use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use *set* instead.

> **Parameters**
>
> - **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application's `root_path`.
>
> - **defaults** – an optional dictionary of default values

from_envvar(*variable_name*, *silent=False*)

Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

> **Parameters**
>
> - **variable_name** – name of the environment variable
>
> - **silent** – set to `True` if you want silent failure for missing files.
>
> **Returns** bool. `True` if able to load config, `False` otherwise.

from_json(*filename*, *silent=False*)

Updates the values in the config from a JSON file. This function behaves as if the JSON object was a dictionary and passed to the `from_mapping()` function.

> **Parameters**
>
> - **filename** – the filename of the JSON file. This can either be an absolute filename or a filename relative to the root path.
>
> - **silent** – set to `True` if you want silent failure for missing files.

New in version 0.11.

from_mapping(**mapping, **kwargs*)

> Updates the config like update() ignoring items with non-upper keys.
>
> New in version 0.11.

from_object(*obj*)

> Updates the values from the given object. An object can be of one of the following two types:
>
> - a string: in this case the object with that name will be imported
>
> - an actual object reference: that object is used directly
>
> Objects are usually either modules or classes. from_object() loads only the uppercase attributes of the module/class. A dict object will not work with from_object() because the keys of a dict are not attributes of the dict class.
>
> Example of module-based configuration:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

> You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with from_pyfile() and ideally from a location not within the package because the package might be installed system wide.
>
> See *Development / Production* for an example of class-based configuration using from_object().
>
> > **Parameters obj** – an import name or object

from_pyfile(*filename, silent=False*)

> Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the from_object() function.
>
> > **Parameters**
> >
> > - **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
> >
> > - **silent** – set to True if you want silent failure for missing files.
>
> New in version 0.7: *silent* parameter.

get_namespace(*namespace, lowercase=True, trim_namespace=True*)

> Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix. Example usage:

```
app.config['IMAGE_STORE_TYPE'] = 'fs'
app.config['IMAGE_STORE_PATH'] = '/var/app/images'
app.config['IMAGE_STORE_BASE_URL'] = 'http://img.website.com'
image_store_config = app.config.get_namespace('IMAGE_STORE_')
```

> The resulting dictionary *image_store_config* would look like:

```
{
    'type': 'fs',
    'path': '/var/app/images',
    'base_url': 'http://img.website.com'
}
```

This is often useful when configuration options map directly to keyword arguments in functions or class constructors.

> **Parameters**
>
> - **namespace** – a configuration namespace
> - **lowercase** – a flag indicating if the keys of the resulting dictionary should be lowercase
> - **trim_namespace** – a flag indicating if the keys of the resulting dictionary should not include the namespace

New in version 0.11.

2.1.15 Stream Helpers

flask.**stream_with_context**(*generator_or_function*)

Request contexts disappear when the response is started on the server. This is done for efficiency reasons and to make it less likely to encounter memory leaks with badly written WSGI middlewares. The downside is that if you are using streamed responses, the generator cannot access request bound information any more.

This function however can help you keep the context around for longer:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    @stream_with_context
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(generate())
```

Alternatively it can also be used around a specific generator:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
```

```
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

New in version 0.9.

2.1.16 Useful Internals

class flask.ctx.**RequestContext**(*app*, *environ*, *request=None*)

The request context contains all request relevant information. It is created at the beginning of the request and pushed to the *_request_ctx_stack* and removed at the end of it. It will create the URL adapter and request object for the WSGI environment provided.

Do not attempt to use this class directly, instead use test_request_context() and request_context() to create this object.

When the request context is popped, it will evaluate all the functions registered on the application for teardown execution (teardown_request()).

The request context is automatically popped at the end of the request for you. In debug mode the request context is kept around if exceptions happen so that interactive debuggers have a chance to introspect the data. With 0.4 this can also be forced for requests that did not fail and outside of DEBUG mode. By setting 'flask._preserve_context' to True on the WSGI environment the context will not pop itself at the end of the request. This is used by the test_client() for example to implement the deferred cleanup functionality.

You might find this helpful for unittests where you need the information from the context local around for a little longer. Make sure to properly pop() the stack yourself in that situation, otherwise your unittests will leak memory.

copy()

Creates a copy of this request context with the same request object. This can be used to move a request context to a different greenlet. Because the actual request object is the same this cannot be used to move a request context to a different thread unless access to the request object is locked.

New in version 0.10.

match_request()

Can be overridden by a subclass to hook into the matching of the request.

pop(*exc=<object object>*)

Pops the request context and unbinds it by doing that. This will also trigger the execution of functions registered by the teardown_request() decorator.

Changed in version 0.9: Added the *exc* argument.

push()

Binds the request context to the current context.

flask.**_request_ctx_stack**

> The internal `LocalStack` that holds RequestContext instances. Typically, the `request` and `session` proxies should be accessed instead of the stack. It may be useful to access the stack in extension code.
>
> 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:

```python
from flask import _request_ctx_stack

def get_session():
    ctx = _request_ctx_stack.top
    if ctx is not None:
        return ctx.session
```

class flask.ctx.**AppContext**(*app*)

> The application context binds an application object implicitly to the current thread or greenlet, similar to how the `RequestContext` binds request information. The application context is also implicitly created if a request context is created but the application is not on top of the individual application context.
>
> **pop**(*exc=<object object>*)
>
> > Pops the app context.
>
> **push**()
>
> > Binds the app context to the current context.

flask.**_app_ctx_stack**

> The internal `LocalStack` that holds AppContext instances. Typically, the `current_app` and g proxies should be accessed instead of the stack. Extensions can access the contexts on the stack as a namespace to store data.
>
> New in version 0.9.

class flask.blueprints.**BlueprintSetupState**(*blueprint,   app,   options, first_registration*)

> Temporary holder object for registering a blueprint with the application. An instance of this class is created by the `make_setup_state()` method and later passed to all register callback functions.
>
> **add_url_rule**(*rule, endpoint=None, view_func=None, **options*)
>
> > A helper method to register a rule (and optionally a view function) to the

application. The endpoint is automatically prefixed with the blueprint's name.

app = None
a reference to the current application

blueprint = None
a reference to the blueprint that created this setup state.

first_registration = None
as blueprints can be registered multiple times with the application and not everything wants to be registered multiple times on it, this attribute can be used to figure out if the blueprint was registered in the past already.

options = None
a dictionary with all options that were passed to the `register_blueprint()` method.

subdomain = None
The subdomain that the blueprint should be active for, `None` otherwise.

url_defaults = None
A dictionary with URL defaults that is added to each and every URL that was defined with the blueprint.

2.1.17 Signals

New in version 0.6.

signals.**signals_available**
`True` if the signaling system is available. This is the case when blinker is installed.

The following signals exist in Flask:

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

Example subscriber:

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

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

flask.**before_render_template**
This signal is sent before template rendering process. The signal is invoked with the instance of the template as *template* and the context as dictionary (named *context*).

Example subscriber:

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

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

flask.**request_started**

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

Example subscriber:

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

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

flask.**request_finished**

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

Example subscriber:

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

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

flask.**got_request_exception**

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

Example subscriber:

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

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

flask.**request_tearing_down**

This signal is sent when the request is tearing down. This is always called, even

if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

```
def close_db_connection(sender, **extra):
    session.close()

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

As of Flask 0.9, this will also be passed an *exc* keyword argument that has a reference to the exception that caused the teardown if there was one.

flask.**appcontext_tearing_down**

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

Example subscriber:

```
def close_db_connection(sender, **extra):
    session.close()

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

This will also be passed an *exc* keyword argument that has a reference to the exception that caused the teardown if there was one.

flask.**appcontext_pushed**

This signal is sent when an application context is pushed. The sender is the application. This is usually useful for unittests in order to temporarily hook in information. For instance it can be used to set a resource early onto the *g* object.

Example usage:

```
from contextlib import contextmanager
from flask import appcontext_pushed

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

And in the testcode:

```
def test_user_me(self):
    with user_set(app, 'john'):
        c = app.test_client()
```

```
        resp = c.get('/users/me')
        assert resp.data == 'username=john'
```

New in version 0.10.

flask.**appcontext_popped**

This signal is sent when an application context is popped. The sender is the application. This usually falls in line with the appcontext_tearing_down signal.

New in version 0.10.

flask.**message_flashed**

This signal is sent when the application is flashing a message. The messages is sent as *message* keyword argument and the category as *category*.

Example subscriber:

```
recorded = []
def record(sender, message, category, **extra):
    recorded.append((message, category))

from flask import message_flashed
message_flashed.connect(record, app)
```

New in version 0.10.

class signals.**Namespace**

An alias for blinker.base.Namespace if blinker is available, otherwise a dummy class that creates fake signals. This class is available for Flask extensions that want to provide the same fallback system as Flask itself.

signal(*name*, *doc=None*)

Creates a new signal for this namespace if blinker is available, otherwise returns a fake signal that has a send method that will do nothing but will fail with a RuntimeError for all other operations, including connecting.

2.1.18 Class-Based Views

New in version 0.7.

class flask.views.**View**

Alternative way to use view functions. A subclass has to implement dispatch_request() which is called with the view arguments from the URL routing system. If methods is provided the methods do not have to be passed to the add_url_rule() method explicitly:

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

    def dispatch_request(self, name):
        return 'Hello %s!' % name
```

```
app.add_url_rule('/hello/<name>', view_func=MyView.as_view('myview'))
```

When you want to decorate a pluggable view you will have to either do that when the view function is created (by wrapping the return value of `as_view()`) or you can use the `decorators` attribute:

```python
class SecretView(View):
    methods = ['GET']
    decorators = [superuser_required]

    def dispatch_request(self):
        ...
```

The decorators stored in the decorators list are applied one after another when the view function is created. Note that you can *not* use the class based decorators since those would decorate the view class and not the generated view function!

classmethod as_view(*name*, **class_args*, ***class_kwargs*)

> Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.
>
> The arguments passed to `as_view()` are forwarded to the constructor of the class.

decorators = ()

> The canonical way to decorate class-based views is to decorate the return value of as_view(). However since this moves parts of the logic from the class declaration to the place where it's hooked into the routing system.
>
> You can place one or more decorators in this list and whenever the view function is created the result is automatically decorated.
>
> New in version 0.8.

dispatch_request()

> Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

methods = None

> A list of methods this view can handle.

provide_automatic_options = None

> Setting this disables or force-enables the automatic options handling.

class flask.views.**MethodView**

> A class-based view that dispatches request methods to the corresponding class methods. For example, if you implement a get method, it will be used to handle GET requests.

```
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'))
```

2.1.19 URL Route Registrations

Generally there are three ways to define rules for the routing system:

1. You can use the `flask.Flask.route()` decorator.

2. You can use the `flask.Flask.add_url_rule()` function.

3. You can directly access the underlying Werkzeug routing system which is exposed as `flask.Flask.url_map`.

Variable parts in the route can be specified with angular brackets (/user/<username>). By default a variable part in the URL accepts any string without a slash however a different converter can be specified as well by using <converter:name>.

Variable parts are passed to the view function as keyword arguments.

The following converters are available:

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

Custom converters can be defined using `flask.Flask.url_map`.

Here are some examples:

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

@app.route('/<username>')
def show_user(username):
    pass

@app.route('/post/<int:post_id>')
```

```
def show_post(post_id):
    pass
```

An important detail to keep in mind is how Flask deals with trailing slashes. The idea is to keep each URL unique so the following rules apply:

1. If a rule ends with a slash and is requested without a slash by the user, the user is automatically redirected to the same page with a trailing slash attached.

2. If a rule does not end with a trailing slash and the user requests the page with a trailing slash, a 404 not found is raised.

This is consistent with how web servers deal with static files. This also makes it possible to use relative link targets safely.

You can also define multiple rules for the same function. They have to be unique however. Defaults can also be specified. Here for example is a definition for a URL that accepts an optional page:

```
@app.route('/users/', defaults={'page': 1})
@app.route('/users/page/<int:page>')
def show_users(page):
    pass
```

This specifies that `/users/` will be the URL for page one and `/users/page/N` will be the URL for page *N*.

Here are the parameters that `route()` and `add_url_rule()` accept. The only difference is that with the route parameter the view function is defined with the decorator instead of the *view_func* parameter.

rule	the URL rule as string
end-point	the endpoint for the registered URL rule. Flask itself assumes that the name of the view function is the name of the endpoint if not explicitly stated.
view_func	the function to call when serving a request to the provided endpoint. If this is not provided one can specify the function later by storing it in the `view_functions` dictionary with the endpoint as key.
de-faults	A dictionary with defaults for this rule. See the example above for how defaults work.
sub-do-main	specifies the rule for the subdomain in case subdomain matching is in use. If not specified the default subdomain is assumed.
***op-tions*	the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling. They have to be specified as keyword arguments.

2.1.20 View Function Options

For internal usage the view functions can have some attributes attached to customize behavior the view function would normally not have control over. The following attributes can be provided optionally to either override some defaults to add_url_rule() or general behavior:

- *__name__*: The name of a function is by default used as endpoint. If endpoint is provided explicitly this value is used. Additionally this will be prefixed with the name of the blueprint by default which cannot be customized from the function itself.

- *methods*: If methods are not provided when the URL rule is added, Flask will look on the view function object itself if a *methods* attribute exists. If it does, it will pull the information for the methods from there.

- *provide_automatic_options*: if this attribute is set Flask will either force enable or disable the automatic implementation of the HTTP OPTIONS response. This can be useful when working with decorators that want to customize the OPTIONS response on a per-view basis.

- *required_methods*: if this attribute is set, Flask will always add these methods when registering a URL rule even if the methods were explicitly overridden in the route() call.

Full example:

```python
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.

2.1.21 Command Line Interface

class flask.cli.**FlaskGroup**(*add_default_commands=True*, *create_app=None*, *add_version_option=True*, *load_dotenv=True*, ***extra*)

Special subclass of the AppGroup group that supports loading more commands from the configured Flask app. Normally a developer does not have to interface with this class but there are some very advanced use cases for which it makes sense to create an instance of this.

For information as of why this is useful see *Custom Scripts*.

Parameters

- **add_default_commands** – if this is True then the default run and shell commands wil be added.

- **add_version_option** – adds the `--version` option.

- **create_app** – an optional callback that is passed the script info and returns the loaded app.

- **load_dotenv** – Load the nearest `.env` and `.flaskenv` files to set environment variables. Will also change the working directory to the directory containing the first file found.

Changed in version 1.0: If installed, python-dotenv will be used to load environment variables from `.env` and `.flaskenv` files.

class `flask.cli.`**AppGroup**(*name=None*, *commands=None*, ***attrs*)

This works similar to a regular click `Group` but it changes the behavior of the `command()` decorator so that it automatically wraps the functions in `with_appcontext()`.

Not to be confused with `FlaskGroup`.

command(**args*, ***kwargs*)

This works exactly like the method of the same name on a regular `click.Group` but it wraps callbacks in `with_appcontext()` unless it's disabled by passing `with_appcontext=False`.

group(**args*, ***kwargs*)

This works exactly like the method of the same name on a regular `click.Group` but it defaults the group class to `AppGroup`.

class `flask.cli.`**ScriptInfo**(*app_import_path=None*, *create_app=None*)

Help object to deal with Flask applications. This is usually not necessary to interface with as it's used internally in the dispatching to click. In future versions of Flask this object will most likely play a bigger role. Typically it's created automatically by the `FlaskGroup` but you can also manually create it and pass it onwards as click object.

app_import_path = None

Optionally the import path for the Flask application.

create_app = None

Optionally a function that is passed the script info to create the instance of the application.

data = None

A dictionary with arbitrary data that can be associated with this script info.

load_app()

Loads the Flask app (if not yet loaded) and returns it. Calling this multiple times will just result in the already loaded app to be returned.

`flask.cli.`**load_dotenv**(*path=None*)

Load "dotenv" files in order of precedence to set environment variables.

If an env var is already set it is not overwritten, so earlier files in the list are preferred over later files.

Changes the current working directory to the location of the first file found, with the assumption that it is in the top level project directory and will be where the Python path should import local packages from.

This is a no-op if python-dotenv is not installed.

> **Parameters path** – Load the file at this location instead of searching.
>
> **Returns** `True` if a file was loaded.

New in version 1.0.

`flask.cli.`**`with_appcontext`**(*f*)

Wraps a callback so that it's guaranteed to be executed with the script's application context. If callbacks are registered directly to the `app.cli` object then they are wrapped with this function by default unless it's disabled.

`flask.cli.`**`pass_script_info`**(*f*)

Marks a function so that an instance of `ScriptInfo` is passed as first argument to the click callback.

`flask.cli.`**`run_command = <click.core.Command object>`**

Run a local development server.

This server is for development purposes only. It does not provide the stability, security, or performance of production WSGI servers.

The reloader and debugger are enabled by default if FLASK_ENV=development or FLASK_DEBUG=1.

`flask.cli.`**`shell_command = <click.core.Command object>`**

Runs an interactive Python shell in the context of a given Flask application. The application will populate the default namespace of this shell according to it's configuration.

This is useful for executing small snippets of management code without having to manually configure the application.

Additional Notes

Design notes, legal information and changelog are here for the interested.

3.1 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.

3.1.1 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:

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


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

Would look like this instead:

```
from hypothetical_flask import route

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

There are three major reasons for this. The most important one is that implicit application objects require that there may only be one instance at the time. There are ways to fake multiple applications with a single application object, like maintaining a stack of applications, but this causes some problems I won't outline here in detail. Now the question is: when does a microframework need more than one application at the same time? A good example for this is unittesting. When you want to test something it can be very helpful to create a minimal application to test specific behavior. When the application object is deleted everything it allocated will be freed again.

Another thing that becomes possible when you have an explicit object lying around in your code is that you can subclass the base class (`Flask`) to alter specific behavior. This would not be possible without hacks if the object were created ahead of time for you based on a class that is not exposed to you.

But there is another very important reason why Flask depends on an explicit instantiation of that class: the package name. Whenever you create a Flask instance you usually pass it *__name__* as package name. Flask depends on that information to properly load resources relative to your module. With Python's outstanding support for reflection it can then access the package to figure out where the templates and static files are stored (see `open_resource()`). Now obviously there are frameworks around that do not need any configuration and will still be able to load templates relative to your application module. But they have to use the current working directory for that, which is a very unreliable way to determine where the application is. The current working directory is process-wide and if you are running multiple applications in one process (which could happen in a webserver without you knowing) the paths will be off. Worse: many webservers do not set the working directory to the directory of your application but to the document root which does not have to be the same folder.

The third reason is "explicit is better than implicit". That object is your WSGI application, you don't have to remember anything else. If you want to apply a WSGI middleware, just wrap it and you're done (though there are better ways to do that so that you do not lose the reference to the application object `wsgi_app()`).

Furthermore this design makes it possible to use a factory function to create the application which is very helpful for unittesting and similar things (*Application Factories*).

3.1.2 The Routing System

Flask uses the Werkzeug routing system which was designed to automatically order routes by complexity. This means that you can declare routes in arbitrary order and they will still work as expected. This is a requirement if you want to properly implement decorator based routing since decorators could be fired in undefined order when the application is split into multiple modules.

Another design decision with the Werkzeug routing system is that routes in Werkzeug try to ensure that URLs are unique. Werkzeug will go quite far with that in that it will automatically redirect to a canonical URL if a route is ambiguous.

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.1.6 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.

3.2 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.

3.2.1 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 (`<br>` becomes `<br/>` or `<br></br>` in XHTML). However, even if the document properly validates as XHTML, what really determines XHTML/HTML processing in browsers is the MIME type, which as said before is often not set properly. So the valid XHTML was being treated as invalid HTML.

XHTML also changed the way JavaScript is used. To properly work with XHTML, programmers have to use the namespaced DOM interface with the XHTML namespace to query for HTML elements.

3.2.2 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 behavior instead of unrealistic and backwards-incompatible specifications.

For example, in HTML4 `<title/Hello/` theoretically parses exactly the same as `<title>Hello</title>`. However, since people were using XHTML-like tags along the lines of `<link />`, browser vendors implemented the XHTML syntax over the syntax defined by the specification.

In 2007, the specification was adopted as the basis of a new HTML specification under the umbrella of the W3C, known as HTML5. Currently, it appears that XHTML is losing traction, as the XHTML 2 working group has been disbanded and HTML5 is being implemented by all major browser vendors.

3.2.3 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
`<tag/value/ == <tag>value</tag>`	✔[1]	✗	✗
` ` supported	✗	✔	✔[2]
`<script/>` supported	✗	✔	✗
should be served as *text/html*	✔	✗[3]	✔
should be served as *application/xhtml+xml*	✗	✔	✗
strict error handling	✗	✔	✗
inline SVG	✗	✔	✔
inline MathML	✗	✔	✔
`<video>` tag	✗	✗	✔
`<audio>` tag	✗	✗	✔
New semantic tags like `<article>`	✗	✗	✔

3.2.4 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.

---

[1] This is an obscure feature inherited from SGML. It is usually not supported by browsers, for reasons detailed above.

[2] This is for compatibility with server code that generates XHTML for tags such as `<br>`. It should not be used in new code.

[3] XHTML 1.0 is the last XHTML standard that allows to be served as *text/html* for backwards compatibility reasons.

- 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:

```html
<!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>
```

3.2.5 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.

3.2.6 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.

3.3 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.

3.3.1 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:

```
<input value="{{ value }}">
```

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 input, 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.

There is one class of XSS issues that Jinja's escaping does not protect against. The a tag's href attribute can contain a *javascript:* URI, which the browser will execute when clicked if not secured properly.

```
<a href="{{ value }}">click here</a>
<a href="javascript:alert('unsafe');">click here</a>
```

To prevent this, you'll need to set the *Content Security Policy (CSP)* response header.

3.3.2 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 have to trick some users to load that page and their profiles will end up being deleted.

Imagine you were to run Facebook with millions of concurrent users and someone would send out links to images of little kittens. When users would go to that page, their profiles would get deleted while they are looking at images of fluffy cats.

How can you prevent that? Basically for each request that modifies content on the server you would have to either use a one-time token and store that in the cookie **and** also transmit it with the form data. After receiving the data on the server again, you would then have to compare the two tokens and ensure they are equal.

Why does Flask not do that for you? The ideal place for this to happen is the form validation framework, which does not exist in Flask.

3.3.3 JSON Security

In Flask 0.10 and lower, `jsonify()` did not serialize top-level arrays to JSON. This was because of a security vulnerability in ECMAScript 4.

ECMAScript 5 closed this vulnerability, so only extremely old browsers are still vulnerable. All of these browsers have other more serious vulnerabilities, so this behavior was changed and `jsonify()` now supports serializing arrays.

3.3.4 Security Headers

Browsers recognize various response headers in order to control security. We recommend reviewing each of the headers below for use in your application. The Flask-Talisman extension can be used to manage HTTPS and the security headers for you.

HTTP Strict Transport Security (HSTS)

Tells the browser to convert all HTTP requests to HTTPS, preventing man-in-the-middle (MITM) attacks.

```
response.headers['Strict-Transport-Security'] = 'max-age=31536000;␣
↪includeSubDomains'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

Content Security Policy (CSP)

Tell the browser where it can load various types of resource from. This header should be used whenever possible, but requires some work to define the correct policy for your site. A very strict policy would be:

```
response.headers['Content-Security-Policy'] = "default-src 'self'"
```

- https://csp.withgoogle.com/docs/index.html

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/
  Content-Security-Policy

X-Content-Type-Options

Forces the browser to honor the response content type instead of trying to detect it, which can be abused to generate a cross-site scripting (XSS) attack.

```
response.headers['X-Content-Type-Options'] = 'nosniff'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/
  X-Content-Type-Options

X-Frame-Options

Prevents external sites from embedding your site in an `iframe`. This prevents a class of attacks where clicks in the outer frame can be translated invisibly to clicks on your page's elements. This is also known as "clickjacking".

```
response.headers['X-Frame-Options'] = 'SAMEORIGIN'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/
  X-Frame-Options

X-XSS-Protection

The browser will try to prevent reflected XSS attacks by not loading the page if the request contains something that looks like JavaScript and the response contains the same data.

```
response.headers['X-XSS-Protection'] = '1; mode=block'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/
  X-XSS-Protection

Set-Cookie options

These options can be added to a `Set-Cookie` header to improve their security. Flask has configuration options to set these on the session cookie. They can be set on other cookies too.

- `Secure` limits cookies to HTTPS traffic only.

- `HttpOnly` protects the contents of cookies from being read with JavaScript.

- `SameSite` restricts how cookies are sent with requests from external sites. Can be set to `'Lax'` (recommended) or `'Strict'`. Lax prevents sending cookies with CSRF-prone requests from external sites, such as submitting a form. `Strict`

prevents sending cookies with all external requests, including following regular links.

```
app.config.update(
    SESSION_COOKIE_SECURE=True,
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SAMESITE='Lax',
)

response.set_cookie('username', 'flask', secure=True, httponly=True, samesite='Lax
↪')
```

Specifying `Expires` or `Max-Age` options, will remove the cookie after the given time, or the current time plus the age, respectively. If neither option is set, the cookie will be removed when the browser is closed.

```
# cookie expires after 10 minutes
response.set_cookie('snakes', '3', max_age=600)
```

For the session cookie, if `session.permanent` is set, then `PERMANENT_SESSION_LIFETIME` is used to set the expiration. Flask's default cookie implementation validates that the cryptographic signature is not older than this value. Lowering this value may help mitigate replay attacks, where intercepted cookies can be sent at a later time.

```
app.config.update(
    PERMANENT_SESSION_LIFETIME=600
)

@app.route('/login', methods=['POST'])
def login():
    ...
    session.clear()
    session['user_id'] = user.id
    session.permanent = True
    ...
```

Use `itsdangerous.TimedSerializer` to sign and validate other cookie values (or any values that need secure signatures).

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

HTTP Public Key Pinning (HPKP)

This tells the browser to authenticate with the server using only the specific certificate key to prevent MITM attacks.

> **Warning:** Be careful when enabling this, as it is very difficult to undo if you set up or upgrade your key incorrectly.

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning

3.4 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.

3.4.1 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.

3.4.2 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.

3.4.3 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:

```python
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:

```python
def write_file(filename, contents, charset='utf-8'):
    with open(filename, 'w') as f:
        f.write(contents.encode(charset))
```

3.4.4 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.

3.5 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.

3.5.1 Anatomy of an Extension

Extensions are all located in a package called `flask_something` where "something" is the name of the library you want to bridge. So for example if you plan to add support for a library named *simplexml* to Flask, you would name your extension's package `flask_simplexml`.

The name of the actual extension (the human readable name) however would be something like "Flask-SimpleXML". Make sure to include the name "Flask" somewhere in that name and that you check the capitalization. This is how users can then register dependencies to your extension in their `setup.py` files.

But what do extensions look like themselves? An extension has to ensure that it works with multiple Flask application instances at once. This is a requirement because many people will use patterns like the *Application Factories* pattern to create their application as needed to aid unittests and to support multiple configurations. Because of that it is crucial that your application supports that kind of behavior.

Most importantly the extension must be shipped with a `setup.py` file and registered on PyPI. Also the development checkout link should work so that people can easily install the development version into their virtualenv without having to download the library by hand.

Flask extensions must be licensed under a BSD, MIT or more liberal license in order to be listed in the Flask Extension Registry. Keep in mind that the Flask Extension Registry is a moderated place and libraries will be reviewed upfront if they behave as required.

3.5.2 "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.

First we create the following folder structure:

```
flask-sqlite3/
    flask_sqlite3.py
    LICENSE
    README
```

Here's the contents of the most important files:

setup.py

The next file that is absolutely required is the `setup.py` file which is used to install your Flask extension. The following contents are something you can work with:

```python
"""
Flask-SQLite3
-------------

This is the description for that library
"""
from setuptools import setup


setup(
    name='Flask-SQLite3',
    version='1.0',
    url='http://example.com/flask-sqlite3/',
    license='BSD',
    author='Your Name',
    author_email='your-email@example.com',
    description='Very short description',
    long_description=__doc__,
    py_modules=['flask_sqlite3'],
    # if you would be using a package instead use packages instead
    # of py_modules:
    # packages=['flask_sqlite3'],
    zip_safe=False,
    include_package_data=True,
    platforms='any',
    install_requires=[
        'Flask'
    ],
    classifiers=[
        'Environment :: Web Environment',
        'Intended Audience :: Developers',
```

```
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        'Topic :: Software Development :: Libraries :: Python Modules'
    ]
)
```

That's a lot of code but you can really just copy/paste that from existing extensions and adapt.

flask_sqlite3.py

Now this is where your extension code goes. But how exactly should such an extension look like? What are the best practices? Continue reading for some insight.

3.5.3 Initializing Extensions

Many extensions will need some kind of initialization step. For example, consider an application that's currently connecting to SQLite like the documentation suggests (*Using SQLite 3 with Flask*). So how does the extension know the name of the application object?

Quite simple: you pass it to it.

There are two recommended ways for an extension to initialize:

initialization functions:

> If your extension is called *helloworld* you might have a function called `init_helloworld(app[, extra_args])` that initializes the extension for that application. It could attach before / after handlers etc.

classes:

> Classes work mostly like initialization functions but can later be used to further change the behavior. For an example look at how the OAuth extension works: there is an *OAuth* object that provides some helper functions like *OAuth.remote_app* to create a reference to a remote application that uses OAuth.

What to use depends on what you have in mind. For the SQLite 3 extension we will use the class-based approach because it will provide users with an object that handles opening and closing database connections.

When designing your classes, it's important to make them easily reusable at the module level. This means the object itself must not under any circumstances store any application specific state and must be shareable between different applications.

3.5.4 The Extension Code

Here's the contents of the *flask_sqlite3.py* for copy/paste:

```python
import sqlite3
from flask import current_app, _app_ctx_stack


class SQLite3(object):
    def __init__(self, app=None):
        self.app = app
        if app is not None:
            self.init_app(app)

    def init_app(self, app):
        app.config.setdefault('SQLITE3_DATABASE', ':memory:')
        app.teardown_appcontext(self.teardown)

    def connect(self):
        return sqlite3.connect(current_app.config['SQLITE3_DATABASE'])

    def teardown(self, exception):
        ctx = _app_ctx_stack.top
        if hasattr(ctx, 'sqlite3_db'):
            ctx.sqlite3_db.close()

    @property
    def connection(self):
        ctx = _app_ctx_stack.top
        if ctx is not None:
            if not hasattr(ctx, 'sqlite3_db'):
                ctx.sqlite3_db = self.connect()
            return ctx.sqlite3_db
```

So here's what these lines of code do:

1. The `__init__` method takes an optional app object and, if supplied, will call `init_app`.

2. The `init_app` method exists so that the `SQLite3` object can be instantiated without requiring an app object. This method supports the factory pattern for creating applications. The `init_app` will set the configuration for the database, defaulting to an in memory database if no configuration is supplied. In addition, the `init_app` method attaches the `teardown` handler.

3. Next, we define a `connect` method that opens a database connection.

4. Finally, we add a `connection` property that on first access opens the database connection and stores it on the context. This is also the recommended way to handling resources: fetch resources lazily the first time they are used.

   Note here that we're attaching our database connection to the top application

context via `_app_ctx_stack.top`. Extensions should use the top context for storing their own information with a sufficiently complex name.

So why did we decide on a class-based approach here? Because using our extension looks something like this:

```python
from flask import Flask
from flask_sqlite3 import SQLite3

app = Flask(__name__)
app.config.from_pyfile('the-config.cfg')
db = SQLite3(app)
```

You can then use the database from views like this:

```python
@app.route('/')
def show_all():
    cur = db.connection.cursor()
    cur.execute(...)
```

Likewise if you are outside of a request you can use the database by pushing an app context:

```python
with app.app_context():
    cur = db.connection.cursor()
    cur.execute(...)
```

At the end of the `with` block the teardown handles will be executed automatically.

Additionally, the `init_app` method is used to support the factory pattern for creating apps:

```python
db = Sqlite3()
# Then later on.
app = create_app('the-config.cfg')
db.init_app(app)
```

Keep in mind that supporting this factory pattern for creating apps is required for approved flask extensions (described below).

Note on `init_app`

As you noticed, `init_app` does not assign app to `self`. This is intentional! Class based Flask extensions must only store the application on the object when the application was passed to the constructor. This tells the extension: I am not interested in using multiple applications.

When the extension needs to find the current application and it does not have a reference to it, it must either use the `current_app` context local or change the API in a way that you can pass the application explicitly.

3.5.5 Using _app_ctx_stack

In the example above, before every request, a `sqlite3_db` variable is assigned to `_app_ctx_stack.top`. In a view function, this variable is accessible using the `connection` property of `SQLite3`. During the teardown of a request, the `sqlite3_db` connection is closed. By using this pattern, the *same* connection to the sqlite3 database is accessible to anything that needs it for the duration of the request.

3.5.6 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 generates useful feedback on what people might want from an extension, but also avoids having multiple developers working in isolation on pretty much the same problem.

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.

3.5.7 Approved Extensions

Flask also has the concept of approved extensions. Approved extensions are tested as part of Flask itself to ensure extensions do not break on new releases. These approved extensions are listed on the Flask Extension Registry and marked appropriately. If you want your own extension to be approved you have to follow these guidelines:

0. An approved Flask extension requires a maintainer. In the event an extension author would like to move beyond the project, the project should find a new maintainer including full source hosting transition and PyPI access. If no maintainer is available, give access to the Flask core team.

1. An approved Flask extension must provide exactly one package or module named `flask_extensionname`.

2. It must ship a testing suite that can either be invoked with `make test` or `python setup.py test`. For test suites invoked with `make test` the extension has to ensure that all dependencies for the test are installed automatically. If tests are invoked with `python setup.py test`, test dependencies can be specified in the `setup.py` file. The test suite also has to be part of the distribution.

3. APIs of approved extensions will be checked for the following characteristics:

- an approved extension has to support multiple applications running in the same Python process.

- it must be possible to use the factory pattern for creating applications.

4. The license must be BSD/MIT/WTFPL licensed.

5. The naming scheme for official extensions is *Flask-ExtensionName* or *ExtensionName-Flask*.

6. Approved extensions must define all their dependencies in the `setup.py` file unless a dependency cannot be met because it is not available on PyPI.

7. The extension must have documentation that uses one of the two Flask themes for Sphinx documentation.

8. The setup.py description (and thus the PyPI description) has to link to the documentation, website (if there is one) and there must be a link to automatically install the development version (`PackageName==dev`).

9. The `zip_safe` flag in the setup script must be set to `False`, even if the extension would be safe for zipping.

10. An extension currently has to support Python 3.4 and newer and 2.7.

3.6 Pocoo Styleguide

The Pocoo styleguide is the styleguide for all Pocoo Projects, including Flask. This styleguide is a requirement for Patches to Flask and a recommendation for Flask extensions.

In general the Pocoo Styleguide closely follows **PEP 8** with some small differences and extensions.

3.6.1 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:

```python
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'
```

3.6.2 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:

```python
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.

3.6.3 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)`

3.6.4 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:

```python
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 a utf-8 encoding declaration (if non ASCII letters are used, but it is recommended all the time) and a standard docstring:

```python
# -*- 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.

3.6.5 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)
```

3.7 Upgrading to Newer Releases

Flask itself is changing like any software is changing over time. Most of the changes are the nice kind, the kind where you don't have to change anything in your code to profit from a new release.

However every once in a while there are changes that do require some changes in your code or there are changes that make it possible for you to improve your own code quality by taking advantage of new features in Flask.

This section of the documentation enumerates all the changes in Flask from release to release and how you can change your code to have a painless updating experience.

Use the **pip** command to upgrade your existing Flask installation by providing the --upgrade parameter:

```
$ pip install --upgrade Flask
```

3.7.1 Version 0.12

Changes to send_file

The filename is no longer automatically inferred from file-like objects. This means that the following code will no longer automatically have X-Sendfile support, etag generation or MIME-type guessing:

```
response = send_file(open('/path/to/file.txt'))
```

Any of the following is functionally equivalent:

```
fname = '/path/to/file.txt'

# Just pass the filepath directly
response = send_file(fname)

# Set the MIME-type and ETag explicitly
response = send_file(open(fname), mimetype='text/plain')
response.set_etag(...)

# Set `attachment_filename` for MIME-type guessing
# ETag still needs to be manually set
```

```
response = send_file(open(fname), attachment_filename=fname)
response.set_etag(...)
```

The reason for this is that some file-like objects have an invalid or even misleading name attribute. Silently swallowing errors in such cases was not a satisfying solution.

Additionally the default of falling back to `application/octet-stream` has been restricted. If Flask can't guess one or the user didn't provide one, the function fails if no filename information was provided.

3.7.2 Version 0.11

0.11 is an odd release in the Flask release cycle because it was supposed to be the 1.0 release. However because there was such a long lead time up to the release we decided to push out a 0.11 release first with some changes removed to make the transition easier. If you have been tracking the master branch which was 1.0 you might see some unexpected changes.

In case you did track the master branch you will notice that **`flask --app`** is removed now. You need to use the environment variable to specify an application.

Debugging

Flask 0.11 removed the `debug_log_format` attribute from Flask applications. Instead the new `LOGGER_HANDLER_POLICY` configuration can be used to disable the default log handlers and custom log handlers can be set up.

Error handling

The behavior of error handlers was changed. The precedence of handlers used to be based on the decoration/call order of `errorhandler()` and `register_error_handler()`, respectively. Now the inheritance hierarchy takes precedence and handlers for more specific exception classes are executed instead of more general ones. See *Error handlers* for specifics.

Trying to register a handler on an instance now raises `ValueError`.

Note: There used to be a logic error allowing you to register handlers only for exception *instances*. This was unintended and plain wrong, and therefore was replaced with the intended behavior of registering handlers only using exception classes and HTTP error codes.

Templating

The `render_template_string()` function has changed to autoescape template variables by default. This better matches the behavior of `render_template()`.

Extension imports

Extension imports of the form `flask.ext.foo` are deprecated, you should use `flask_foo`.

The old form still works, but Flask will issue a `flask.exthook.ExtDeprecationWarning` for each extension you import the old way. We also provide a migration utility called flask-ext-migrate that is supposed to automatically rewrite your imports for this.

3.7.3 Version 0.10

The biggest change going from 0.9 to 0.10 is that the cookie serialization format changed from pickle to a specialized JSON format. This change has been done in order to avoid the damage an attacker can do if the secret key is leaked. When you upgrade you will notice two major changes: all sessions that were issued before the upgrade are invalidated and you can only store a limited amount of types in the session. The new sessions are by design much more restricted to only allow JSON with a few small extensions for tuples and strings with HTML markup.

In order to not break people's sessions it is possible to continue using the old session system by using the Flask-OldSessions extension.

Flask also started storing the `flask.g` object on the application context instead of the request context. This change should be transparent for you but it means that you now can store things on the g object when there is no request context yet but an application context. The old `flask.Flask.request_globals_class` attribute was renamed to `flask.Flask.app_ctx_globals_class`.

3.7.4 Version 0.9

The behavior of returning tuples from a function was simplified. If you return a tuple it no longer defines the arguments for the response object you're creating, it's now always a tuple in the form (`response`, `status`, `headers`) where at least one item has to be provided. If you depend on the old behavior, you can add it easily by subclassing Flask:

```python
class TraditionalFlask(Flask):
    def make_response(self, rv):
        if isinstance(rv, tuple):
            return self.response_class(*rv)
        return Flask.make_response(self, rv)
```

If you maintain an extension that was using _request_ctx_stack before, please consider changing to _app_ctx_stack if it makes sense for your extension. For instance, the app context stack makes sense for extensions which connect to databases. Using the app context stack instead of the request context stack will make extensions more readily handle use cases outside of requests.

3.7.5 Version 0.8

Flask introduced a new session interface system. We also noticed that there was a naming collision between flask.session the module that implements sessions and flask.session which is the global session object. With that introduction we moved the implementation details for the session system into a new module called flask.sessions. If you used the previously undocumented session support we urge you to upgrade.

If invalid JSON data was submitted Flask will now raise a BadRequest exception instead of letting the default ValueError bubble up. This has the advantage that you no longer have to handle that error to avoid an internal server error showing up for the user. If you were catching this down explicitly in the past as ValueError you will need to change this.

Due to a bug in the test client Flask 0.7 did not trigger teardown handlers when the test client was used in a with statement. This was since fixed but might require some changes in your test suites if you relied on this behavior.

3.7.6 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 a unified diff with changes it assumes are safe to apply. However as this is an automated tool it won't be able to find all use cases and it might miss some. We internally spread a lot of deprecation warnings all over the place to make it easy to find pieces of code that it was unable to upgrade.

We strongly recommend that you hand review the generated patchfile and only apply the chunks that look good.

If you are using git as version control system for your project we recommend applying the patch with path -p1 < patchfile.diff and then using the interactive commit feature to only apply the chunks that look good.

To apply the upgrade script do the following:

1. Download the script: flask-07-upgrade.py

2. Run it in the directory of your application:

```
python flask-07-upgrade.py > patchfile.diff
```

3. Review the generated patchfile.

4. Apply the patch:

```
patch -p1 < patchfile.diff
```

5. If you were using per-module template folders you need to move some templates around. Previously if you had a folder named `templates` next to a blueprint named `admin` the implicit template path automatically was `admin/index.html` for a template file called `templates/index.html`. This no longer is the case. Now you need to name the template `templates/admin/index.html`. The tool will not detect this so you will have to do that on your own.

Please note that deprecation warnings are disabled by default starting with Python 2.7. In order to see the deprecation warnings that might be emitted you have to enabled them with the `warnings` module.

If you are working with windows and you lack the `patch` command line utility you can get it as part of various Unix runtime environments for windows including cygwin, msysgit or ming32. Also source control systems like svn, hg or git have builtin support for applying unified diffs as generated by the tool. Check the manual of your version control system for more information.

Bug in Request Locals

Due to a bug in earlier implementations the request local proxies now raise a `RuntimeError` instead of an `AttributeError` when they are unbound. If you caught these exceptions with `AttributeError` before, you should catch them with `RuntimeError` now.

Additionally the `send_file()` function is now issuing deprecation warnings if you depend on functionality that will be removed in Flask 0.11. Previously it was possible to use etags and mimetypes when file objects were passed. This was unreliable and caused issues for a few setups. If you get a deprecation warning, make sure to update your application to work with either filenames there or disable etag attaching and attach them yourself.

Old code:

```
return send_file(my_file_object)
return send_file(my_file_object)
```

New code:

```
return send_file(my_file_object, add_etags=False)
```

Upgrading to new Teardown Handling

We streamlined the behavior of the callbacks for request handling. For things that modify the response the after_request() decorators continue to work as expected, but for things that absolutely must happen at the end of request we introduced the new teardown_request() decorator. Unfortunately that change also made after-request work differently under error conditions. It's not consistently skipped if exceptions happen whereas previously it might have been called twice to ensure it is executed at the end of the request.

If you have database connection code that looks like this:

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

You are now encouraged to use this instead:

```python
@app.teardown_request
def after_request(exception):
    if hasattr(g, 'db'):
        g.db.close()
```

On the upside this change greatly improves the internal code flow and makes it easier to customize the dispatching and error handling. This makes it now a lot easier to write unit tests as you can prevent closing down of database connections for a while. You can take advantage of the fact that the teardown callbacks are called when the response context is removed from the stack so a test can query the database after request handling:

```python
with app.test_client() as client:
    resp = client.get('/')
    # g.db is still bound if there is such a thing

# and here it's gone
```

Manual Error Handler Attaching

While it is still possible to attach error handlers to Flask.error_handlers it's discouraged to do so and in fact deprecated. In general we no longer recommend custom error handler attaching via assignments to the underlying dictionary due to the more complex internal handling to support arbitrary exception classes and blueprints. See Flask.errorhandler() for more information.

The proper upgrade is to change this:

```python
app.error_handlers[403] = handle_error
```

Into this:

```
app.register_error_handler(403, handle_error)
```

Alternatively you should just attach the function with a decorator:

```
@app.errorhandler(403)
def handle_error(e):
    ...
```

(Note that `register_error_handler()` is new in Flask 0.7)

Blueprint Support

Blueprints replace the previous concept of "Modules" in Flask. They provide better semantics for various features and work better with large applications. The update script provided should be able to upgrade your applications automatically, but there might be some cases where it fails to upgrade. What changed?

- Blueprints need explicit names. Modules had an automatic name guessing scheme where the shortname for the module was taken from the last part of the import module. The upgrade script tries to guess that name but it might fail as this information could change at runtime.

- Blueprints have an inverse behavior for `url_for()`. Previously `.foo` told `url_for()` that it should look for the endpoint `foo` on the application. Now it means "relative to current module". The script will inverse all calls to `url_for()` automatically for you. It will do this in a very eager way so you might end up with some unnecessary leading dots in your code if you're not using modules.

- Blueprints do not automatically provide static folders. They will also no longer automatically export templates from a folder called `templates` next to their location however but it can be enabled from the constructor. Same with static files: if you want to continue serving static files you need to tell the constructor explicitly the path to the static folder (which can be relative to the blueprint's module path).

- Rendering templates was simplified. Now the blueprints can provide template folders which are added to a general template searchpath. This means that you need to add another subfolder with the blueprint's name into that folder if you want `blueprintname/template.html` as the template name.

If you continue to use the `Module` object which is deprecated, Flask will restore the previous behavior as good as possible. However we strongly recommend upgrading to the new blueprints as they provide a lot of useful improvement such as the ability to attach a blueprint multiple times, blueprint specific error handlers and a lot more.

3.7.7 Version 0.6

Flask 0.6 comes with a backwards incompatible change which affects the order of after-request handlers. Previously they were called in the order of the registration, now

they are called in reverse order. This change was made so that Flask behaves more like people expected it to work and how other systems handle request pre- and post-processing. If you depend on the order of execution of post-request functions, be sure to change the order.

Another change that breaks backwards compatibility is that context processors will no longer override values passed directly to the template rendering function. If for example `request` is as variable passed directly to the template, the default context processor will not override it with the current request object. This makes it easier to extend context processors later to inject additional variables without breaking existing template not expecting them.

3.7.8 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.

3.7.9 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.

3.7.10 Version 0.3

Flask 0.3 introduces configuration support and logging as well as categories for flashing messages. All these are features that are 100% backwards compatible but you might want to take advantage of them.

Configuration Support

The configuration support makes it easier to write any kind of application that requires some sort of configuration. (Which most likely is the case for any application out there).

If you previously had code like this:

```
app.debug = DEBUG
app.secret_key = SECRET_KEY
```

You no longer have to do that, instead you can just load a configuration into the config object. How this works is outlined in *Configuration Handling*.

Logging Integration

Flask now configures a logger for you with some basic and useful defaults. If you run your application in production and want to profit from automatic error logging, you might be interested in attaching a proper log handler. Also you can start logging warnings and errors into the logger when appropriately. For more information on that, read *Application Errors*.

Categories for Flash Messages

Flash messages can now have categories attached. This makes it possible to render errors, warnings or regular messages differently for example. This is an opt-in feature because it requires some rethinking in the code.

Read all about that in the *Message Flashing* pattern.

3.8 Flask Changelog

3.8.1 Version 1.0

unreleased

- **Python 2.6 and 3.3 are no longer supported.** (pallets/meta#24)

- Bump minimum dependency versions to the latest stable versions: Werkzeug >= 0.14, Jinja >= 2.10, itsdangerous >= 0.24, Click >= 5.1. (#2586)

- Skip `app.run` when a Flask application is run from the command line. This avoids some behavior that was confusing to debug.

- Change the default for `JSONIFY_PRETTYPRINT_REGULAR` to `False`. `jsonify()` returns a compact format by default, and an indented format in debug mode. (#2193)

- `Flask.__init__` accepts the `host_matching` argument and sets it on `url_map`. (#1559)

- `Flask.__init__` accepts the `static_host` argument and passes it as the host argument when defining the static route. (#1559)

- `send_file()` supports Unicode in `attachment_filename`. (#2223)

- Pass _scheme argument from `url_for()` to `handle_url_build_error()`. (#2017)

- `add_url_rule()` accepts the `provide_automatic_options` argument to disable adding the `OPTIONS` method. (#1489)

- `MethodView` subclasses inherit method handlers from base classes. (#1936)

- Errors caused while opening the session at the beginning of the request are handled by the app's error handlers. (#2254)

- Blueprints gained `json_encoder` and `json_decoder` attributes to override the app's encoder and decoder. (#1898)

- `Flask.make_response()` raises `TypeError` instead of `ValueError` for bad response types. The error messages have been improved to describe why the type is invalid. (#2256)

- Add `routes` CLI command to output routes registered on the application. (#2259)

- Show warning when session cookie domain is a bare hostname or an IP address, as these may not behave properly in some browsers, such as Chrome. (#2282)

- Allow IP address as exact session cookie domain. (#2282)

- `SESSION_COOKIE_DOMAIN` is set if it is detected through `SERVER_NAME`. (#2282)

- Auto-detect zero-argument app factory called `create_app` or `make_app` from `FLASK_APP`. (#2297)

- Factory functions are not required to take a `script_info` parameter to work with the `flask` command. If they take a single parameter or a parameter named `script_info`, the `ScriptInfo` object will be passed. (#2319)

- `FLASK_APP` can be set to an app factory, with arguments if needed, for example `FLASK_APP=myproject.app:create_app('dev')`. (#2326)

- `FLASK_APP` can point to local packages that are not installed in editable mode, although `pip install -e` is still preferred. (#2414)

- The `View` class attribute `provide_automatic_options` is set in `as_view()`, to be detected by `add_url_rule()`. (#2316)

- Error handling will try handlers registered for `blueprint, code, app, code, blueprint, exception, app, exception`. (#2314)

- `Cookie` is added to the response's `Vary` header if the session is accessed at all during the request (and not deleted). (#2288)

- `test_request_context()` accepts `subdomain` and `url_scheme` arguments for use when building the base URL. (#1621)

- Set `APPLICATION_ROOT` to `'/'` by default. This was already the implicit default when it was set to `None`.

- `TRAP_BAD_REQUEST_ERRORS` is enabled by default in debug mode. `BadRequestKeyError` has a message with the bad key in debug mode instead of the generic bad request message. (#2348)

- Allow registering new tags with `TaggedJSONSerializer` to support storing other types in the session cookie. (#2352)

- Only open the session if the request has not been pushed onto the context stack yet. This allows `stream_with_context()` generators to access the same session that the containing view uses. (#2354)

- Add `json` keyword argument for the test client request methods. This will dump the given object as JSON and set the appropriate content type. (#2358)

- Extract JSON handling to a mixin applied to both the `Request` and `Response` classes. This adds the `is_json()` and `get_json()` methods to the response to make testing JSON response much easier. (#2358)

- Removed error handler caching because it caused unexpected results for some exception inheritance hierarchies. Register handlers explicitly for each exception if you want to avoid traversing the MRO. (#2362)

- Fix incorrect JSON encoding of aware, non-UTC datetimes. (#2374)

- Template auto reloading will honor debug mode even even if `jinja_env` was already accessed. (#2373)

- The following old deprecated code was removed. (#2385)

    - `flask.ext` - import extensions directly by their name instead of through the `flask.ext` namespace. For example, `import flask.ext.sqlalchemy` becomes `import flask_sqlalchemy`.

    - `Flask.init_jinja_globals` - extend `Flask.create_jinja_environment()` instead.

    - `Flask.error_handlers` - tracked by `Flask.error_handler_spec`, use `Flask.errorhandler()` to register handlers.

    - `Flask.request_globals_class` - use `Flask.app_ctx_globals_class` instead.

    - `Flask.static_path` - use `Flask.static_url_path` instead.

    - `Request.module` - use `Request.blueprint` instead.

- The `Request.json` property is no longer deprecated. (#1421)

- Support passing a `EnvironBuilder` or dict to `test_client.open`. (#2412)

- The `flask` command and `Flask.run()` will load environment variables from `.env` and `.flaskenv` files if python-dotenv is installed. (#2416)

- When passing a full URL to the test client, the scheme in the URL is used instead of `PREFERRED_URL_SCHEME`. (#2430)

- `Flask.logger` has been simplified. `LOGGER_NAME` and `LOGGER_HANDLER_POLICY` config was removed. The logger is always named `flask.app`. The level is only set on first access, it doesn't check `Flask.debug` each time. Only one format is used, not different ones depending on `Flask.debug`. No handlers are removed, and a handler is only added if no handlers are already configured. (#2436)

- Blueprint view function names may not contain dots. (#2450)

- Fix a `ValueError` caused by invalid `Range` requests in some cases. (#2526)

- The development server uses threads by default. (#2529)

- Loading config files with `silent=True` will ignore `ENOTDIR` errors. (#2581)

- Pass `--cert` and `--key` options to `flask run` to run the development server over HTTPS. (#2606)

- Added `SESSION_COOKIE_SAMESITE` to control the `SameSite` attribute on the session cookie. (#2607)

- Added `test_cli_runner()` to create a Click runner that can invoke Flask CLI commands for testing. (#2636)

- Subdomain matching is disabled by default and setting `SERVER_NAME` does not implicitly enable it. It can be enabled by passing `subdomain_matching=True` to the `Flask` constructor. (#2635)

- A single trailing slash is stripped from the blueprint `url_prefix` when it is registered with the app. (#2629)

- `Request.get_json()` doesn't cache the result if parsing fails when `silent` is true. (#2651)

- `Request.get_json()` no longer accepts arbitrary encodings. Incoming JSON should be encoded using UTF-8 per **RFC 8259**, but Flask will autodetect UTF-8, -16, or -32. (#2691)

- Added `MAX_COOKIE_SIZE` and `Response.max_cookie_size` to control when Werkzeug warns about large cookies that browsers may ignore. (#2693)

3.8.2 Version 0.12.2

Released on May 16 2017

- Fix a bug in *safe_join* on Windows.

3.8.3 Version 0.12.1

Bugfix release, released on March 31st 2017

- Prevent *flask run* from showing a NoAppException when an ImportError occurs within the imported application module.

- Fix encoding behavior of `app.config.from_pyfile` for Python 3. Fix #2118.

- Use the `SERVER_NAME` config if it is present as default values for `app.run`. #2109, #2152
- Call *ctx.auto_pop* with the exception object instead of *None*, in the event that a *BaseException* such as *KeyboardInterrupt* is raised in a request handler.

3.8.4 Version 0.12

Released on December 21st 2016, codename Punsch.

- the cli command now responds to *–version*.
- Mimetype guessing and ETag generation for file-like objects in `send_file` has been removed, as per issue #104. See pull request #1849.
- Mimetype guessing in `send_file` now fails loudly and doesn't fall back to `application/octet-stream`. See pull request #1988.
- Make `flask.safe_join` able to join multiple paths like `os.path.join` (pull request #1730).
- Revert a behavior change that made the dev server crash instead of returning a Internal Server Error (pull request #2006).
- Correctly invoke response handlers for both regular request dispatching as well as error handlers.
- Disable logger propagation by default for the app logger.
- Add support for range requests in `send_file`.
- `app.test_client` includes preset default environment, which can now be directly set, instead of per `client.get`.

3.8.5 Version 0.11.2

Bugfix release, unreleased

- Fix crash when running under PyPy3, see pull request #1814.

3.8.6 Version 0.11.1

Bugfix release, released on June 7th 2016.

- Fixed a bug that prevented `FLASK_APP=foobar/__init__.py` from working. See pull request #1872.

3.8.7 Version 0.11

Released on May 29th 2016, codename Absinthe.

- Added support to serializing top-level arrays to `flask.jsonify()`. This introduces a security risk in ancient browsers. See *JSON Security* for details.

- Added before_render_template signal.

- Added ***kwargs* to `flask.Test.test_client()` to support passing additional keyword arguments to the constructor of `flask.Flask.test_client_class`.

- Added `SESSION_REFRESH_EACH_REQUEST` config key that controls the set-cookie behavior. If set to `True` a permanent session will be refreshed each request and get their lifetime extended, if set to `False` it will only be modified if the session actually modifies. Non permanent sessions are not affected by this and will always expire if the browser window closes.

- Made Flask support custom JSON mimetypes for incoming data.

- Added support for returning tuples in the form (`response,  headers`) from a view function.

- Added `flask.Config.from_json()`.

- Added `flask.Flask.config_class`.

- Added `flask.Config.get_namespace()`.

- Templates are no longer automatically reloaded outside of debug mode. This can be configured with the new `TEMPLATES_AUTO_RELOAD` config key.

- Added a workaround for a limitation in Python 3.3's namespace loader.

- Added support for explicit root paths when using Python 3.3's namespace packages.

- Added **flask** and the `flask.cli` module to start the local debug server through the click CLI system. This is recommended over the old `flask.run()` method as it works faster and more reliable due to a different design and also replaces `Flask-Script`.

- Error handlers that match specific classes are now checked first, thereby allowing catching exceptions that are subclasses of HTTP exceptions (in `werkzeug. exceptions`). This makes it possible for an extension author to create exceptions that will by default result in the HTTP error of their choosing, but may be caught with a custom error handler if desired.

- Added `flask.Config.from_mapping()`.

- Flask will now log by default even if debug is disabled. The log format is now hardcoded but the default log handling can be disabled through the `LOGGER_HANDLER_POLICY` configuration key.

- Removed deprecated module functionality.

- Added the `EXPLAIN_TEMPLATE_LOADING` config flag which when enabled will instruct Flask to explain how it locates templates. This should help users debug when the wrong templates are loaded.

- Enforce blueprint handling in the order they were registered for template loading.

- Ported test suite to py.test.

- Deprecated `request.json` in favour of `request.get_json()`.

- Add "pretty" and "compressed" separators definitions in jsonify() method. Reduces JSON response size when JSONIFY_PRETTYPRINT_REGULAR=False by removing unnecessary white space included by default after separators.

- JSON responses are now terminated with a newline character, because it is a convention that UNIX text files end with a newline and some clients don't deal well when this newline is missing. See https://github.com/pallets/flask/pull/1262 – this came up originally as a part of https://github.com/kennethreitz/httpbin/issues/168

- The automatically provided `OPTIONS` method is now correctly disabled if the user registered an overriding rule with the lowercase-version `options` (issue #1288).

- `flask.json.jsonify` now supports the `datetime.date` type (pull request #1326).

- Don't leak exception info of already catched exceptions to context teardown handlers (pull request #1393).

- Allow custom Jinja environment subclasses (pull request #1422).

- Updated extension dev guidelines.

- `flask.g` now has `pop()` and `setdefault` methods.

- Turn on autoescape for `flask.templating.render_template_string` by default (pull request #1515).

- `flask.ext` is now deprecated (pull request #1484).

- `send_from_directory` now raises BadRequest if the filename is invalid on the server OS (pull request #1763).

- Added the `JSONIFY_MIMETYPE` configuration variable (pull request #1728).

- Exceptions during teardown handling will no longer leave bad application contexts lingering around.

3.8.8 Version 0.10.2

(bugfix release, release date to be announced)

- Fixed broken *test_appcontext_signals()* test case.

- Raise an `AttributeError` in `flask.helpers.find_package()` with a useful message explaining why it is raised when a PEP 302 import hook is used without an *is_package()* method.

- Fixed an issue causing exceptions raised before entering a request or app context to be passed to teardown handlers.

- Fixed an issue with query parameters getting removed from requests in the test client when absolute URLs were requested.

- Made *@before_first_request* into a decorator as intended.

- Fixed an etags bug when sending a file streams with a name.

- Fixed *send_from_directory* not expanding to the application root path correctly.

- Changed logic of before first request handlers to flip the flag after invoking. This will allow some uses that are potentially dangerous but should probably be permitted.

- Fixed Python 3 bug when a handler from *app.url_build_error_handlers* reraises the *BuildError*.

3.8.9 Version 0.10.1

(bugfix release, released on June 14th 2013)

- Fixed an issue where |tojson was not quoting single quotes which made the filter not work properly in HTML attributes. Now it's possible to use that filter in single quoted attributes. This should make using that filter with angular.js easier.

- Added support for byte strings back to the session system. This broke compatibility with the common case of people putting binary data for token verification into the session.

- Fixed an issue where registering the same method twice for the same endpoint would trigger an exception incorrectly.

3.8.10 Version 0.10

Released on June 13th 2013, codename Limoncello.

- Changed default cookie serialization format from pickle to JSON to limit the impact an attacker can do if the secret key leaks. See *Version 0.10* for more information.

- Added `template_test` methods in addition to the already existing `template_filter` method family.

- Added `template_global` methods in addition to the already existing `template_filter` method family.

- Set the content-length header for x-sendfile.

- `tojson` filter now does not escape script blocks in HTML5 parsers.

- `tojson` used in templates is now safe by default due. This was allowed due to the different escaping behavior.

- Flask will now raise an error if you attempt to register a new function on an already used endpoint.

- Added wrapper module around simplejson and added default serialization of datetime objects. This allows much easier customization of how JSON is handled by Flask or any Flask extension.

- Removed deprecated internal `flask.session` module alias. Use `flask.sessions` instead to get the session module. This is not to be confused with `flask.session` the session proxy.

- Templates can now be rendered without request context. The behavior is slightly different as the `request`, `session` and `g` objects will not be available and blueprint's context processors are not called.

- The config object is now available to the template as a real global and not through a context processor which makes it available even in imported templates by default.

- Added an option to generate non-ascii encoded JSON which should result in less bytes being transmitted over the network. It's disabled by default to not cause confusion with existing libraries that might expect `flask.json.dumps` to return bytestrings by default.

- `flask.g` is now stored on the app context instead of the request context.

- `flask.g` now gained a `get()` method for not erroring out on non existing items.

- `flask.g` now can be used with the `in` operator to see what's defined and it now is iterable and will yield all attributes stored.

- `flask.Flask.request_globals_class` got renamed to `flask.Flask.app_ctx_globals_class` which is a better name to what it does since 0.10.

- *request*, *session* and *g* are now also added as proxies to the template context which makes them available in imported templates. One has to be very careful with those though because usage outside of macros might cause caching.

- Flask will no longer invoke the wrong error handlers if a proxy exception is passed through.

- Added a workaround for chrome's cookies in localhost not working as intended with domain names.

- Changed logic for picking defaults for cookie values from sessions to work better with Google Chrome.

- Added *message_flashed* signal that simplifies flashing testing.

- Added support for copying of request contexts for better working with greenlets.

- Removed custom JSON HTTP exception subclasses. If you were relying on them you can reintroduce them again yourself trivially. Using them however is strongly discouraged as the interface was flawed.

- Python requirements changed: requiring Python 2.6 or 2.7 now to prepare for Python 3.3 port.

- Changed how the teardown system is informed about exceptions. This is now more reliable in case something handles an exception halfway through the error handling process.

- Request context preservation in debug mode now keeps the exception information around which means that teardown handlers are able to distinguish error from success cases.

- Added the `JSONIFY_PRETTYPRINT_REGULAR` configuration variable.

- Flask now orders JSON keys by default to not trash HTTP caches due to different hash seeds between different workers.

- Added *appcontext_pushed* and *appcontext_popped* signals.

- The builtin run method now takes the `SERVER_NAME` into account when picking the default port to run on.

- Added *flask.request.get_json()* as a replacement for the old *flask.request.json* property.

3.8.11 Version 0.9

Released on July 1st 2012, codename Campari.

- The `flask.Request.on_json_loading_failed()` now returns a JSON formatted response by default.

- The `flask.url_for()` function now can generate anchors to the generated links.

- The `flask.url_for()` function now can also explicitly generate URL rules specific to a given HTTP method.

- Logger now only returns the debug log setting if it was not set explicitly.

- Unregister a circular dependency between the WSGI environment and the request object when shutting down the request. This means that environ `werkzeug.request` will be `None` after the response was returned to the WSGI server but has the advantage that the garbage collector is not needed on CPython to tear down the request unless the user created circular dependencies themselves.

- Session is now stored after callbacks so that if the session payload is stored in the session you can still modify it in an after request callback.

- The `flask.Flask` class will avoid importing the provided import name if it can (the required first parameter), to benefit tools which build Flask instances programmatically. The Flask class will fall back to using import on systems with custom module hooks, e.g. Google App Engine, or when the import name is inside a zip archive (usually a .egg) prior to Python 2.7.

- Blueprints now have a decorator to add custom template filters application wide, `flask.Blueprint.app_template_filter()`.

- The Flask and Blueprint classes now have a non-decorator method for adding custom template filters application wide, `flask.Flask.add_template_filter()` and `flask.Blueprint.add_app_template_filter()`.

- The `flask.get_flashed_messages()` function now allows rendering flashed message categories in separate blocks, through a `category_filter` argument.

- The `flask.Flask.run()` method now accepts `None` for *host* and *port* arguments, using default values when `None`. This allows for calling run using configuration values, e.g. `app.run(app.config.get('MYHOST'), app.config.get('MYPORT'))`, with proper behavior whether or not a config file is provided.

- The `flask.render_template()` method now accepts a either an iterable of template names or a single template name. Previously, it only accepted a single template name. On an iterable, the first template found is rendered.

- Added `flask.Flask.app_context()` which works very similar to the request context but only provides access to the current application. This also adds support for URL generation without an active request context.

- View functions can now return a tuple with the first instance being an instance of `flask.Response`. This allows for returning `jsonify(error="error msg"), 400` from a view function.

- `Flask` and `Blueprint` now provide a `get_send_file_max_age()` hook for subclasses to override behavior of serving static files from Flask when using `flask.Flask.send_static_file()` (used for the default static file handler) and `send_file()`. This hook is provided a filename, which for example allows changing cache controls by file extension. The default max-age for *send_file* and static files can be configured through a new `SEND_FILE_MAX_AGE_DEFAULT` configuration variable, which is used in the default *get_send_file_max_age* implementation.

- Fixed an assumption in sessions implementation which could break message flashing on sessions implementations which use external storage.

- Changed the behavior of tuple return values from functions. They are no longer arguments to the response object, they now have a defined meaning.

- Added `flask.Flask.request_globals_class` to allow a specific class to be used on creation of the g instance of each request.

- Added *required_methods* attribute to view functions to force-add methods on registration.

- Added `flask.after_this_request()`.

- Added `flask.stream_with_context()` and the ability to push contexts multiple times without producing unexpected behavior.

3.8.12 Version 0.8.1

Bugfix release, released on July 1st 2012

- Fixed an issue with the undocumented *flask.session* module to not work properly on Python 2.5. It should not be used but did cause some problems for package managers.

3.8.13 Version 0.8

Released on September 29th 2011, codename Rakija

- Refactored session support into a session interface so that the implementation of the sessions can be changed without having to override the Flask class.

- Empty session cookies are now deleted properly automatically.

- View functions can now opt out of getting the automatic OPTIONS implementation.

- HTTP exceptions and Bad Request errors can now be trapped so that they show up normally in the traceback.

- Flask in debug mode is now detecting some common problems and tries to warn you about them.

- Flask in debug mode will now complain with an assertion error if a view was attached after the first request was handled. This gives earlier feedback when users forget to import view code ahead of time.

- Added the ability to register callbacks that are only triggered once at the beginning of the first request. (`Flask.before_first_request()`)

- Malformed JSON data will now trigger a bad request HTTP exception instead of a value error which usually would result in a 500 internal server error if not handled. This is a backwards incompatible change.

- Applications now not only have a root path where the resources and modules are located but also an instance path which is the designated place to drop files that are modified at runtime (uploads etc.). Also this is conceptually only instance depending and outside version control so it's the perfect place to put configuration files etc. For more information see *Instance Folders*.

- Added the `APPLICATION_ROOT` configuration variable.

- Implemented `session_transaction()` to easily modify sessions from the test environment.

- Refactored test client internally. The `APPLICATION_ROOT` configuration variable as well as `SERVER_NAME` are now properly used by the test client as defaults.

- Added `flask.views.View.decorators` to support simpler decorating of pluggable (class-based) views.

- Fixed an issue where the test client if used with the "with" statement did not trigger the execution of the teardown handlers.

- Added finer control over the session cookie parameters.

- HEAD requests to a method view now automatically dispatch to the *get* method if no handler was implemented.

- Implemented the virtual `flask.ext` package to import extensions from.

- The context preservation on exceptions is now an integral component of Flask itself and no longer of the test client. This cleaned up some internal logic and lowers the odds of runaway request contexts in unittests.

3.8.14 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.

3.8.15 Version 0.7.2

Bugfix release, released on July 6th 2011

- Fixed an issue with URL processors not properly working on blueprints.

3.8.16 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.

3.8.17 Version 0.7

Released on June 28th 2011, codename Grappa

- Added `make_default_options_response()` which can be used by subclasses to alter the default behavior for `OPTIONS` responses.

- Unbound locals now raise a proper `RuntimeError` instead of an `AttributeError`.

- Mimetype guessing and etag support based on file objects is now deprecated for `flask.send_file()` because it was unreliable. Pass filenames instead or attach your own etags and provide a proper mimetype by hand.

- Static file handling for modules now requires the name of the static folder to be supplied explicitly. The previous autodetection was not reliable and caused issues on Google's App Engine. Until 1.0 the old behavior will continue to work but issue dependency warnings.

- fixed a problem for Flask to run on jython.

- added a `PROPAGATE_EXCEPTIONS` configuration variable that can be used to flip the setting of exception propagation which previously was linked to `DEBUG` alone and is now linked to either `DEBUG` or `TESTING`.

- Flask no longer internally depends on rules being added through the *add_url_rule* function and can now also accept regular werkzeug rules added to the url map.

- Added an *endpoint* method to the flask application object which allows one to register a callback to an arbitrary endpoint with a decorator.

- Use Last-Modified for static file sending instead of Date which was incorrectly introduced in 0.6.

- Added *create_jinja_loader* to override the loader creation process.

- Implemented a silent flag for *config.from_pyfile*.

- Added *teardown_request* decorator, for functions that should run at the end of a request regardless of whether an exception occurred. Also the behavior for *after_request* was changed. It's now no longer executed when an exception is raised. See *Upgrading to new Teardown Handling*

- Implemented `flask.has_request_context()`

- Deprecated *init_jinja_globals*. Override the `create_jinja_environment()` method instead to achieve the same functionality.

- Added `flask.safe_join()`

- The automatic JSON request data unpacking now looks at the charset mimetype parameter.

- Don't modify the session on `flask.get_flashed_messages()` if there are no messages in the session.

- *before_request* handlers are now able to abort requests with errors.

- it is not possible to define user exception handlers. That way you can provide custom error messages from a central hub for certain errors that might occur during request processing (for instance database connection errors, timeouts from remote resources etc.).

- Blueprints can provide blueprint specific error handlers.

- Implemented generic *Pluggable Views* (class-based views).

3.8.18 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.

3.8.19 Version 0.6

Released on July 27th 2010, codename Whisky

- after request functions are now called in reverse order of registration.

- OPTIONS is now automatically implemented by Flask unless the application explicitly adds 'OPTIONS' as method to the URL rule. In this case no automatic OPTIONS handling kicks in.

- static rules are now even in place if there is no static folder for the module. This was implemented to aid GAE which will remove the static folder if it's part of a mapping in the .yml file.

- the `config` is now available in the templates as *config*.

- context processors will no longer override values passed directly to the render function.

- added the ability to limit the incoming request data with the new `MAX_CONTENT_LENGTH` configuration value.

- the endpoint for the `flask.Module.add_url_rule()` method is now optional to be consistent with the function of the same name on the application object.

- added a `flask.make_response()` function that simplifies creating response object instances in views.

- added signalling support based on blinker. This feature is currently optional and supposed to be used by extensions and applications. If you want to use it, make sure to have blinker installed.

- refactored the way URL adapters are created. This process is now fully customizable with the `create_url_adapter()` method.

- modules can now register for a subdomain instead of just an URL prefix. This makes it possible to bind a whole module to a configurable subdomain.

3.8.20 Version 0.5.2

Bugfix Release, released on July 15th 2010

- fixed another issue with loading templates from directories when modules were used.

3.8.21 Version 0.5.1

Bugfix Release, released on July 6th 2010

- fixes an issue with template loading from directories when modules where used.

3.8.22 Version 0.5

Released on July 6th 2010, codename Calvados

- fixed a bug with subdomains that was caused by the inability to specify the server name. The server name can now be set with the SERVER_NAME config key. This key is now also used to set the session cookie cross-subdomain wide.

- autoescaping is no longer active for all templates. Instead it is only active for `.html`, `.htm`, `.xml` and `.xhtml`. Inside templates this behavior can be changed with the `autoescape` tag.

- refactored Flask internally. It now consists of more than a single file.

- `flask.send_file()` now emits etags and has the ability to do conditional responses builtin.

- (temporarily) dropped support for zipped applications. This was a rarely used feature and led to some confusing behavior.

- added support for per-package template and static-file directories.

- removed support for *create_jinja_loader* which is no longer used in 0.5 due to the improved module support.

- added a helper function to expose files from any directory.

3.8.23 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.

3.8.24 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)

3.8.25 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.

3.8.26 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.

3.8.27 Version 0.1

First public preview release.

3.9 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.

3.9.1 Authors

Flask is developed and maintained by the Pallets team and community contributors. It was created by Armin Ronacher. The core maintainers are:

- David Lord (davidism)
- Adrian Mönnich (ThiefMaster)
- Armin Ronacher (mitsuhiko)
- Marcus Unterwaditzer (untitaker)

A full list of contributors is available from git with:

```
git shortlog -sne
```

3.9.2 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 source code shipped as part of Flask (Flask itself as well as the examples and the unittests) as well as documentation.
- The *"Flask Artwork License"* applies to the project's Horn-Logo.

3.9.3 Flask License

Copyright © 2010 by the Pallets team.

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.

- Neither the name of the copyright holder nor the names of its contributors may 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 HOLDER 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.

3.9.4 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.

3.10 How to contribute to Flask

Thank you for considering contributing to Flask!

3.10.1 Support questions

Please, don't use the issue tracker for this. Use one of the following resources for questions about your own code:

- The IRC channel #pocoo on FreeNode.

- The IRC channel #python on FreeNode for more general questions.

- The mailing list flask@python.org for long term discussion or larger issues.

- Ask on Stack Overflow. Search with Google first using: `site:stackoverflow.com flask {search term, exception message, etc.}`

3.10.2 Reporting issues

- Describe what you expected to happen.

- If possible, include a minimal, complete, and verifiable example to help us identify the issue. This also helps check that the issue is not with your own code.

- Describe what actually happened. Include the full traceback if there was an exception.

- List your Python, Flask, and Werkzeug versions. If possible, check if this issue is already fixed in the repository.

3.10.3 Submitting patches

- Include tests if your patch is supposed to solve a bug, and explain clearly under which circumstances the bug happens. Make sure the test fails without your patch.

- Try to follow PEP8, but you may ignore the line length limit if following it would make the code uglier.

First time setup

- Download and install the latest version of git.

- Configure git with your username and email:

```
git config --global user.name 'your name'
git config --global user.email 'your email'
```

- Make sure you have a GitHub account.

- Fork Flask to your GitHub account by clicking the Fork button.

- Clone your GitHub fork locally:

```
git clone https://github.com/{username}/flask
cd flask
```

- Add the main repository as a remote to update later:

```
git remote add pallets https://github.com/pallets/flask
git fetch pallets
```

- Create a virtualenv:

```
python3 -m venv env
. env/bin/activate
# or "env\Scripts\activate" on Windows
```

- Install Flask in editable mode with development dependencies:

```
pip install -e ".[dev]"
```

Start coding

- Create a branch to identify the issue you would like to work on (e.g. 2287-dry-test-suite)

- Using your favorite editor, make your changes, committing as you go.

- Try to follow PEP8, but you may ignore the line length limit if following it would make the code uglier.

- Include tests that cover any code changes you make. Make sure the test fails without your patch. *Run the tests..*

- Push your commits to GitHub and create a pull request.

- Celebrate

Running the tests

Run the basic test suite with:

```
pytest
```

This only runs the tests for the current environment. Whether this is relevant depends on which part of Flask you're working on. Travis-CI will run the full suite when you submit your pull request.

The full test suite takes a long time to run because it tests multiple combinations of Python and dependencies. You need to have Python 2.7, 3.4, 3.5 3.6, and PyPy 2.7 installed to run all of the environments. Then run:

```
tox
```

Running test coverage

Generating a report of lines that do not have test coverage can indicate where to start contributing. Run `pytest` using `coverage` and generate a report on the terminal and as an interactive HTML document:

```
coverage run -m pytest
coverage report
coverage html
# then open htmlcov/index.html
```

Read more about coverage.

Running the full test suite with `tox` will combine the coverage reports from all runs.

Building the docs

Build the docs in the docs directory using Sphinx:

```
cd docs
make html
```

Open `_build/html/index.html` in your browser to view the docs.

Read more about Sphinx.

make targets

Flask provides a `Makefile` with various shortcuts. They will ensure that all dependencies are installed.

- `make test` runs the basic test suite with `pytest`
- `make cov` runs the basic test suite with `coverage`

- `make test-all` runs the full test suite with `tox`

- `make docs` builds the HTML documentation

3.10.4 Caution: zero-padded file modes

This repository contains several zero-padded file modes that may cause issues when pushing this repository to git hosts other than GitHub. Fixing this is destructive to the commit history, so we suggest ignoring these warnings. If it fails to push and you're using a self-hosted git service like GitLab, you can turn off repository checks in the admin panel.

These files can also cause issues while cloning. If you have

```
[fetch]
fsckobjects = true
```

or

```
[receive]
fsckObjects = true
```

set in your git configuration file, cloning this repository will fail. The only solution is to set both of the above settings to false while cloning, and then setting them back to true after the cloning is finished.

Python Module Index

f

flask, 197
flask.json, 253
flask.json.tag, 255

Index