Werkzeug Documentation

Release 0.11.dev

May 03, 2015

CONTENTS

Part I

GETTING STARTED

If you are new to Werkzeug or WSGI development in general you should start here.

CHAPTER

ONE

INSTALLATION

Werkzeug requires at least Python 2.6 to work correctly. If you do need to support an older version you can download an older version of Werkzeug though we strongly recommend against that. Werkzeug currently has experimental support for Python 3. For more information about the Python 3 support see *Python 3 Notes*.

1.1 Installing a released version

1.1.1 As a Python egg (via easy_install or pip)

You can install the most recent Werkzeug version using easy_install:

```
easy_install Werkzeug
```

Alternatively you can also use pip:

```
pip install Werkzeug
```

Either way we strongly recommend using these tools in combination with *virtualenv*.

This will install a Werkzeug egg in your Python installation's *site-packages* directory.

1.1.2 From the tarball release

1. Download the most recent tarball from the download page.
2. Unpack the tarball.
3. `python setup.py install`

Note that the last command will automatically download and install setuptools if you don't already have it installed. This requires a working Internet connection.

This will install Werkzeug into your Python installation's *site-packages* directory.

1.2 Installing the development version

1. Install Git
2. `git clone git://github.com/mitsuhiko/werkzeug.git`
3. `cd werkzeug`
4. `pip install --editable .`

1.3 virtualenv

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

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

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

So let's see how virtualenv works!

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

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

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

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

If you are on Windows and don't have the *easy_install* command, you must install it first. Once you have it installed, run the same commands as above, but without the *sudo* prefix.

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

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

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

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

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

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

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

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

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

```
$ pip install Werkzeug
```

A few seconds later you are good to go.

CHAPTER

TWO

TRANSITION TO WERKZEUG 1.0

Werkzeug originally had a magical import system hook that enabled everything to be imported from one module and still loading the actual implementations lazily as necessary. Unfortunately this turned out to be slow and also unreliable on alternative Python implementations and Google's App Engine.

Starting with 0.7 we recommend against the short imports and strongly encourage starting importing from the actual implementation module. Werkzeug 1.0 will disable the magical import hook completely.

Because finding out where the actual functions are imported and rewriting them by hand is a painful and boring process we wrote a tool that aids in making this transition.

2.1 Automatically Rewriting Imports

For instance, with Werkzeug < 0.7 the recommended way to use the escape function was this:

```
from werkzeug import escape
```

With Werkzeug 0.7, the recommended way to import this function is directly from the utils module (and with 1.0 this will become mandatory). To automatically rewrite all imports one can use the werkzeug-import-rewrite script.

You can use it by executing it with Python and with a list of folders with Werkzeug based code. It will then spit out a hg/git compatible patch file. Example patch file creation:

```
$ python werkzeug-import-rewrite.py . > new-imports.udiff
```

To apply the patch one of the following methods work:

hg:

```
    hg import new-imports.udiff
```

git:

```
git apply new-imports.udiff
```

patch:

```
patch -p1 < new-imports.udiff
```

2.2 Stop Using Deprecated Things

A few things in Werkzeug will stop being supported and for others, we're suggesting alternatives even if they will stick around for a longer time.

Do not use:

- *werkzeug.script*, replace it with custom scripts written with *argparse* or something similar.
- *werkzeug.template*, replace with a proper template engine.
- *werkzeug.contrib.jsrouting*, stop using URL generation for JavaScript, it does not scale well with many public routing.
- *werkzeug.contrib.kickstart*, replace with hand written code, the Werkzeug API became better in general that this is no longer necessary.
- *werkzeug.contrib.testtools*, not useful really.

CHAPTER

THREE

WERKZEUG TUTORIAL

Welcome to the Werkzeug tutorial in which we will create a TinyURL clone that stores URLs in a redis instance. The libraries we will use for this applications are Jinja 2 for the templates, redis for the database layer and, of course, Werkzeug for the WSGI layer.

You can use *pip* to install the required libraries:

```
pip install Jinja2 redis
```

Also make sure to have a redis server running on your local machine. If you are on OS X, you can use *brew* to install it:

```
brew install redis
```

If you are on Ubuntu or Debian, you can use apt-get:

```
sudo apt-get install redis-server
```

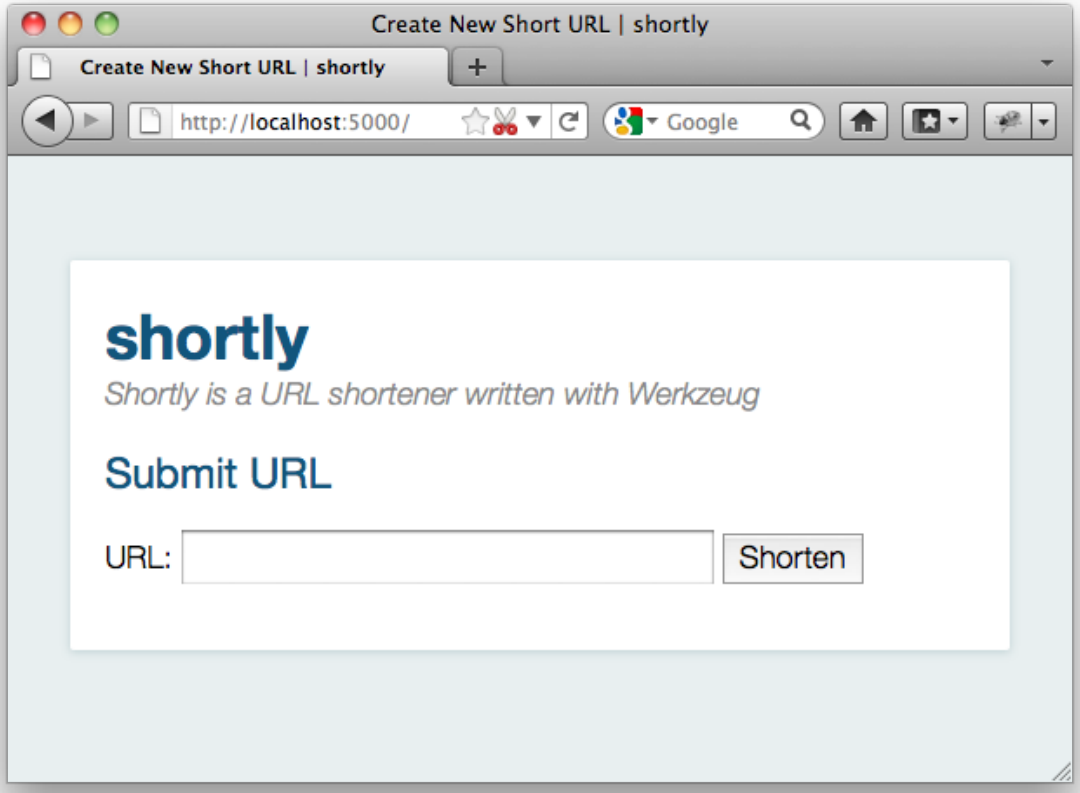
Redis was developed for UNIX systems and was never really designed to work on Windows. For development purposes, the unofficial ports however work well enough. You can get them from github.

3.1 Introducing Shortly

In this tutorial, we will together create a simple URL shortener service with Werkzeug. Please keep in mind that Werkzeug is not a framework, it's a library with utilities to create your own framework or application and as such is very flexible. The approach we use here is just one of many you can use.

As data store, we will use redis here instead of a relational database to keep this simple and because that's the kind of job that redis excels at.

The final result will look something like this:



3.2 Step 0: A Basic WSGI Introduction

Werkzeug is a utility library for WSGI. WSGI itself is a protocol or convention that ensures that your web application can speak with the webserver and more importantly that web applications work nicely together.

A basic "Hello World" application in WSGI without the help of Werkzeug looks like this:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello World!']
```

A WSGI application is something you can call and pass an environ dict and a `start_response` callable. The environ contains all incoming information, the `start_response` function can be used to indicate the start of the response. With Werkzeug you don't have to deal directly with either as request and response objects are provided to work with them.

The request data takes the environ object and allows you to access the data from that environ in a nice manner. The response object is a WSGI application in itself and provides a much nicer way to create responses.

Here is how you would write that application with response objects:

```
from werkzeug.wrappers import Response

def application(environ, start_response):
    response = Response('Hello World!', mimetype='text/plain')
    return response(environ, start_response)
```

And here an expanded version that looks at the query string in the URL (more importantly at the *name* parameter in the URL to substitute "World" against another word):

```
from werkzeug.wrappers import Request, Response

def application(environ, start_response):
    request = Request(environ)
    text = 'Hello %s!' % request.args.get('name', 'World')
    response = Response(text, mimetype='text/plain')
    return response(environ, start_response)
```

And that's all you need to know about WSGI.

3.3 Step 1: Creating the Folders

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

```
/shortly
    /static
    /templates
```

The shortly folder is not a python package, but just something where we drop our files. Directly into this folder we will then put our main module in the following steps. The files inside the static folder are available to users of the application via HTTP. This is the place where CSS and JavaScript files go. Inside the templates folder we will make Jinja2 look for templates. The templates you create later in the tutorial will go in this directory.

3.4 Step 2: The Base Structure

Now let's get right into it and create a module for our application. Let's create a file called *shortly.py* in the *shortly* folder. At first we will need a bunch of imports. I will pull in all the imports here, even if they are not used right away, to keep it from being confusing:

```
import os
import redis
import urlparse
from werkzeug.wrappers import Request, Response
from werkzeug.routing import Map, Rule
from werkzeug.exceptions import HTTPException, NotFound
```

```
from werkzeug.wsgi import SharedDataMiddleware
from werkzeug.utils import redirect
from jinja2 import Environment, FileSystemLoader
```

Then we can create the basic structure for our application and a function to create a new instance of it, optionally with a piece of WSGI middleware that exports all the files on the *static* folder on the web:

```
class Shortly(object):

    def __init__(self, config):
        self.redis = redis.Redis(config['redis_host'], config['redis_port'])

    def dispatch_request(self, request):
        return Response('Hello World!')

    def wsgi_app(self, environ, start_response):
        request = Request(environ)
        response = self.dispatch_request(request)
        return response(environ, start_response)

    def __call__(self, environ, start_response):
        return self.wsgi_app(environ, start_response)


def create_app(redis_host='localhost', redis_port=6379, with_static=True):
    app = Shortly({
        'redis_host':       redis_host,
        'redis_port':       redis_port
    })
    if with_static:
        app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
            '/static':  os.path.join(os.path.dirname(__file__), 'static')
        })
    return app
```

Lastly we can add a piece of code that will start a local development server with automatic code reloading and a debugger:

```
if __name__ == '__main__':
    from werkzeug.serving import run_simple
    app = create_app()
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

The basic idea here is that our `Shortly` class is an actual WSGI application. The `__call__` method directly dispatches to `wsgi_app`. This is done so that we can wrap `wsgi_app` to apply middlewares like we do in the `create_app` function. The actual `wsgi_app` method then creates a `Request` object and calls the `dispatch_request` method which then has to return a `Response` object which is then evaluated as WSGI application again. As you can see: turtles all the way down. Both the `Shortly` class we create, as well as any request object in Werkzeug implements the WSGI inter-

face. As a result of that you could even return another WSGI application from the `dispatch_request` method.

The `create_app` factory function can be used to create a new instance of our application. Not only will it pass some parameters as configuration to the application but also optionally add a WSGI middleware that exports static files. This way we have access to the files from the static folder even when we are not configuring our server to provide them which is very helpful for development.

3.5 Intermezzo: Running the Application

Now you should be able to execute the file with *python* and see a server on your local machine:

```
$ python shortly.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader: stat() polling
```

It also tells you that the reloader is active. It will use various techniques to figure out if any file changed on the disk and then automatically restart.

Just go to the URL and you should see "Hello World!".

3.6 Step 3: The Environment

Now that we have the basic application class, we can make the constructor do something useful and provide a few helpers on there that can come in handy. We will need to be able to render templates and connect to redis, so let's extend the class a bit:

```
def __init__(self, config):
    self.redis = redis.Redis(config['redis_host'], config['redis_port'])
    template_path = os.path.join(os.path.dirname(__file__), 'templates')
    self.jinja_env = Environment(loader=FileSystemLoader(template_path),
                                 autoescape=True)

def render_template(self, template_name, **context):
    t = self.jinja_env.get_template(template_name)
    return Response(t.render(context), mimetype='text/html')
```

3.7 Step 4: The Routing

Next up is routing. Routing is the process of matching and parsing the URL to something we can use. Werkzeug provides a flexible integrated routing system which we can use for that. The way it works is that you create a *`Map`* instance and add a bunch of *`Rule`* objects. Each rule has a pattern it will try to match the URL against and an

"endpoint". The endpoint is typically a string and can be used to uniquely identify the URL. We could also use this to automatically reverse the URL, but that's not what we will do in this tutorial.

Just put this into the constructor:

```
self.url_map = Map([
    Rule('/', endpoint='new_url'),
    Rule('/<short_id>', endpoint='follow_short_link'),
    Rule('/<short_id>+', endpoint='short_link_details')
])
```

Here we create a URL map with three rules. / for the root of the URL space where we will just dispatch to a function that implements the logic to create a new URL. And then one that follows the short link to the target URL and another one with the same rule but a plus (+) at the end to show the link details.

So how do we find our way from the endpoint to a function? That's up to you. The way we will do it in this tutorial is by calling the method `on_` + endpoint on the class itself. Here is how this works:

```
def dispatch_request(self, request):
    adapter = self.url_map.bind_to_environ(request.environ)
    try:
        endpoint, values = adapter.match()
        return getattr(self, 'on_' + endpoint)(request, **values)
    except HTTPException, e:
        return e
```

We bind the URL map to the current environment and get back a `URLAdapter`. The adapter can be used to match the request but also to reverse URLs. The match method will return the endpoint and a dictionary of values in the URL. For instance the rule for `follow_short_link` has a variable part called `short_id`. When we go to `http://localhost:5000/foo` we will get the following values back:

```
endpoint = 'follow_short_link'
values = {'short_id': u'foo'}
```

If it does not match anything, it will raise a *NotFound* exception, which is an *HTTPException*. All HTTP exceptions are also WSGI applications by themselves which render a default error page. So we just catch all of them down and return the error itself.

If all works well, we call the function `on_` + endpoint and pass it the request as argument as well as all the URL arguments as keyword arguments and return the response object that method returns.

3.8 Step 5: The First View

Let's start with the first view: the one for new URLs:

```
def on_new_url(self, request):
    error = None
    url = ''
    if request.method == 'POST':
        url = request.form['url']
        if not is_valid_url(url):
            error = 'Please enter a valid URL'
        else:
            short_id = self.insert_url(url)
            return redirect('/%s+' % short_id)
    return self.render_template('new_url.html', error=error, url=url)
```

This logic should be easy to understand. Basically we are checking that the request method is POST, in which case we validate the URL and add a new entry to the database, then redirect to the detail page. This means we need to write a function and a helper method. For URL validation this is good enough:

```
def is_valid_url(url):
    parts = urlparse.urlparse(url)
    return parts.scheme in ('http', 'https')
```

For inserting the URL, all we need is this little method on our class:

```
def insert_url(self, url):
    short_id = self.redis.get('reverse-url:' + url)
    if short_id is not None:
        return short_id
    url_num = self.redis.incr('last-url-id')
    short_id = base36_encode(url_num)
    self.redis.set('url-target:' + short_id, url)
    self.redis.set('reverse-url:' + url, short_id)
    return short_id
```

`reverse-url:` + the URL will store the short id. If the URL was already submitted this won't be None and we can just return that value which will be the short ID. Otherwise we increment the `last-url-id` key and convert it to base36. Then we store the link and the reverse entry in redis. And here the function to convert to base 36:

```
def base36_encode(number):
    assert number >= 0, 'positive integer required'
    if number == 0:
        return '0'
    base36 = []
    while number != 0:
        number, i = divmod(number, 36)
        base36.append('0123456789abcdefghijklmnopqrstuvwxyz'[i])
    return ''.join(reversed(base36))
```

So what is missing for this view to work is the template. We will create this later, let's first also write the other views and then do the templates in one go.

3.9 Step 6: Redirect View

The redirect view is easy. All it has to do is to look for the link in redis and redirect to it. Additionally we will also increment a counter so that we know how often a link was clicked:

```
def on_follow_short_link(self, request, short_id):
    link_target = self.redis.get('url-target:' + short_id)
    if link_target is None:
        raise NotFound()
    self.redis.incr('click-count:' + short_id)
    return redirect(link_target)
```

In this case we will raise a *NotFound* exception by hand if the URL does not exist, which will bubble up to the `dispatch_request` function and be converted into a default 404 response.

3.10 Step 7: Detail View

The link detail view is very similar, we just render a template again. In addition to looking up the target, we also ask redis for the number of times the link was clicked and let it default to zero if such a key does not yet exist:

```
def on_short_link_details(self, request, short_id):
    link_target = self.redis.get('url-target:' + short_id)
    if link_target is None:
        raise NotFound()
    click_count = int(self.redis.get('click-count:' + short_id) or 0)
    return self.render_template('short_link_details.html',
        link_target=link_target,
        short_id=short_id,
        click_count=click_count
    )
```

Please be aware that redis always works with strings, so you have to convert the click count to `int` by hand.

3.11 Step 8: Templates

And here are all the templates. Just drop them into the *templates* folder. Jinja2 supports template inheritance, so the first thing we will do is create a layout template with blocks that act as placeholders. We also set up Jinja2 so that it automatically escapes strings with HTML rules, so we don't have to spend time on that ourselves. This prevents XSS attacks and rendering errors.

layout.html:

```
<!doctype html>
<title>{% block title %}{% endblock %} | shortly</title>
<link rel=stylesheet href=/static/style.css type=text/css>
<div class=box>
  <h1><a href=/>shortly</a></h1>
  <p class=tagline>Shortly is a URL shortener written with Werkzeug
  {% block body %}{% endblock %}
</div>
```

new_url.html:

```
{% extends "layout.html" %}
{% block title %}Create New Short URL{% endblock %}
{% block body %}
  <h2>Submit URL</h2>
  <form action="" method=post>
    {% if error %}
      <p class=error><strong>Error:</strong> {{ error }}
    {% endif %}
    <p>URL:
      <input type=text name=url value="{{ url }}" class=urlinput>
      <input type=submit value="Shorten">
  </form>
{% endblock %}
```

short_link_details.html:

```
{% extends "layout.html" %}
{% block title %}Details about /{{ short_id }}{% endblock %}
{% block body %}
  <h2><a href="/{{ short_id }}">/{{ short_id }}</a></h2>
  <dl>
    <dt>Full link
    <dd class=link><div>{{ link_target }}</div>
    <dt>Click count:
    <dd>{{ click_count }}
  </dl>
{% endblock %}
```

3.12 Step 9: The Style

For this to look better than ugly black and white, here a simple stylesheet that goes along:

```
body        { background: #E8EFF0; margin: 0; padding: 0; }
body, input { font-family: 'Helvetica Neue', Arial,
              sans-serif; font-weight: 300; font-size: 18px; }
.box        { width: 500px; margin: 60px auto; padding: 20px;
              background: white; box-shadow: 0 1px 4px #BED1D4;
```

```
                border-radius: 2px; }
a             { color: #11557C; }
h1, h2        { margin: 0; color: #11557C; }
h1 a          { text-decoration: none; }
h2            { font-weight: normal; font-size: 24px; }
.tagline      { color: #888; font-style: italic; margin: 0 0 20px 0; }
.link div     { overflow: auto; font-size: 0.8em; white-space: pre;
                padding: 4px 10px; margin: 5px 0; background: #E5EAF1; }
dt            { font-weight: normal; }
.error        { background: #E8EFF0; padding: 3px 8px; color: #11557C;
                font-size: 0.9em; border-radius: 2px; }
.urlinput     { width: 300px; }
```

3.13 Bonus: Refinements

Look at the implementation in the example dictionary in the Werkzeug repository to see a version of this tutorial with some small refinements such as a custom 404 page.

- shortly in the example folder

CHAPTER

FOUR

API LEVELS

Werkzeug is intended to be a utility rather than a framework. Because of that the user-friendly API is separated from the lower-level API so that Werkzeug can easily be used to extend another system.

All the functionality the `Request` and `Response` objects (aka the "wrappers") provide is also available in small utility functions.

4.1 Example

This example implements a small *Hello World* application that greets the user with the name entered:

```
from werkzeug.utils import escape
from werkzeug.wrappers import Request, Response

@Request.application
def hello_world(request):
    result = ['<title>Greeter</title>']
    if request.method == 'POST':
        result.append('<h1>Hello %s!</h1>' % escape(request.form['name']))
    result.append('''
        <form action="" method="post">
            <p>Name: <input type="text" name="name" size="20">
            <input type="submit" value="Greet me">
        </form>
    ''')
    return Response(''.join(result), mimetype='text/html')
```

Alternatively the same application could be used without request and response objects but by taking advantage of the parsing functions werkzeug provides:

```
from werkzeug.formparser import parse_form_data
from werkzeug.utils import escape

def hello_world(environ, start_response):
    result = ['<title>Greeter</title>']
```

```
    if environ['REQUEST_METHOD'] == 'POST':
        form = parse_form_data(environ)[1]
        result.append('<h1>Hello %s!</h1>' % escape(form['name']))
    result.append('''
        <form action="" method="post">
            <p>Name: <input type="text" name="name" size="20">
            <input type="submit" value="Greet me">
        </form>
    ''')
    start_response('200 OK', [('Content-Type', 'text/html; charset=utf-8')])
    return [''.join(result)]
```

4.2 High or Low?

Usually you want to use the high-level layer (the request and response objects). But there are situations where this might not be what you want.

For example you might be maintaining code for an application written in Django or another framework and you have to parse HTTP headers. You can utilize Werkzeug for that by accessing the lower-level HTTP header parsing functions.

Another situation where the low level parsing functions can be useful are custom WSGI frameworks, unit-testing or modernizing an old CGI/mod_python application to WSGI as well as WSGI middlewares where you want to keep the overhead low.

CHAPTER

FIVE

QUICKSTART

This part of the documentation shows how to use the most important parts of Werkzeug. It's intended as a starting point for developers with basic understanding of **PEP 333** (WSGI) and **RFC 2616** (HTTP).

> **Warning:** Make sure to import all objects from the places the documentation suggests. It is theoretically possible in some situations to import objects from different locations but this is not supported.
> For example `MultiDict` is a member of the *werkzeug* module but internally implemented in a different one.

5.1 WSGI Environment

The WSGI environment contains all the information the user request transmits to the application. It is passed to the WSGI application but you can also create a WSGI environ dict using the `create_environ()` helper:

```
>>> from werkzeug.test import create_environ
>>> environ = create_environ('/foo', 'http://localhost:8080/')
```

Now we have an environment to play around:

```
>>> environ['PATH_INFO']
'/foo'
>>> environ['SCRIPT_NAME']
''
>>> environ['SERVER_NAME']
'localhost'
```

Usually nobody wants to work with the environ directly because it is limited to bytestrings and does not provide any way to access the form data besides parsing that data by hand.

5.2 Enter Request

For access to the request data the `Request` object is much more fun. It wraps the *environ* and provides a read-only access to the data from there:

```
>>> from werkzeug.wrappers import Request
>>> request = Request(environ)
```

Now you can access the important variables and Werkzeug will parse them for you and decode them where it makes sense. The default charset for requests is set to *utf-8* but you can change that by subclassing `Request`.

```
>>> request.path
u'/foo'
>>> request.script_root
u''
>>> request.host
'localhost:8080'
>>> request.url
'http://localhost:8080/foo'
```

We can also find out which HTTP method was used for the request:

```
>>> request.method
'GET'
```

This way we can also access URL arguments (the query string) and data that was transmitted in a POST/PUT request.

For testing purposes we can create a request object from supplied data using the `from_values()` method:

```
>>> from cStringIO import StringIO
>>> data = "name=this+is+encoded+form+data&another_key=another+one"
>>> request = Request.from_values(query_string='foo=bar&blah=blafasel',
...    content_length=len(data), input_stream=StringIO(data),
...    content_type='application/x-www-form-urlencoded',
...    method='POST')
...
>>> request.method
'POST'
```

Now we can access the URL parameters easily:

```
>>> request.args.keys()
['blah', 'foo']
>>> request.args['blah']
u'blafasel'
```

Same for the supplied form data:

```
>>> request.form['name']
u'this is encoded form data'
```

Handling for uploaded files is not much harder as you can see from this example:

```
def store_file(request):
    file = request.files.get('my_file')
    if file:
        file.save('/where/to/store/the/file.txt')
    else:
        handle_the_error()
```

The files are represented as `FileStorage` objects which provide some common operations to work with them.

Request headers can be accessed by using the `headers` attribute:

```
>>> request.headers['Content-Length']
'54'
>>> request.headers['Content-Type']
'application/x-www-form-urlencoded'
```

The keys for the headers are of course case insensitive.

5.3 Header Parsing

There is more. Werkzeug provides convenient access to often used HTTP headers and other request data.

Let's create a request object with all the data a typical web browser transmits so that we can play with it:

```
>>> environ = create_environ()
>>> environ.update(
...     HTTP_USER_AGENT='Mozilla/5.0 (Macintosh; U; Mac OS X 10.5; en-US; ) Firefox/3.1',
...     HTTP_ACCEPT='text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
...     HTTP_ACCEPT_LANGUAGE='de-at,en-us;q=0.8,en;q=0.5',
...     HTTP_ACCEPT_ENCODING='gzip,deflate',
...     HTTP_ACCEPT_CHARSET='ISO-8859-1,utf-8;q=0.7,*;q=0.7',
...     HTTP_IF_MODIFIED_SINCE='Fri, 20 Feb 2009 10:10:25 GMT',
...     HTTP_IF_NONE_MATCH='"e51c9-1e5d-46356dc86c640"',
...     HTTP_CACHE_CONTROL='max-age=0'
... )
...
>>> request = Request(environ)
```

Let's start with the most useless header: the user agent:

```
>>> request.user_agent.browser
'firefox'
>>> request.user_agent.platform
'macos'
>>> request.user_agent.version
'3.1'
```

```
>>> request.user_agent.language
'en-US'
```

A more useful header is the accept header. With this header the browser informs the web application what mimetypes it can handle and how well. All accept headers are sorted by the quality, the best item being the first:

```
>>> request.accept_mimetypes.best
'text/html'
>>> 'application/xhtml+xml' in request.accept_mimetypes
True
>>> print request.accept_mimetypes["application/json"]
0.8
```

The same works for languages:

```
>>> request.accept_languages.best
'de-at'
>>> request.accept_languages.values()
['de-at', 'en-us', 'en']
```

And of course encodings and charsets:

```
>>> 'gzip' in request.accept_encodings
True
>>> request.accept_charsets.best
'ISO-8859-1'
>>> 'utf-8' in request.accept_charsets
True
```

Normalization is available, so you can safely use alternative forms to perform containment checking:

```
>>> 'UTF8' in request.accept_charsets
True
>>> 'de_AT' in request.accept_languages
True
```

E-tags and other conditional headers are available in parsed form as well:

```
>>> request.if_modified_since
datetime.datetime(2009, 2, 20, 10, 10, 25)
>>> request.if_none_match
<ETags '"e51c9-1e5d-46356dc86c640"'>
>>> request.cache_control
<RequestCacheControl 'max-age=0'>
>>> request.cache_control.max_age
0
>>> 'e51c9-1e5d-46356dc86c640' in request.if_none_match
True
```

5.4 Responses

Response objects are the opposite of request objects. They are used to send data back to the client. In reality, response objects are nothing more than glorified WSGI applications.

So what you are doing is not *returning* the response objects from your WSGI application but *calling* it as WSGI application inside your WSGI application and returning the return value of that call.

So imagine your standard WSGI "Hello World" application:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello World!']
```

With response objects it would look like this:

```
from werkzeug.wrappers import Response

def application(environ, start_response):
    response = Response('Hello World!')
    return response(environ, start_response)
```

Also, unlike request objects, response objects are designed to be modified. So here is what you can do with them:

```
>>> from werkzeug.wrappers import Response
>>> response = Response("Hello World!")
>>> response.headers['content-type']
'text/plain; charset=utf-8'
>>> response.data
'Hello World!'
>>> response.headers['content-length'] = len(response.data)
```

You can modify the status of the response in the same way. Either just the code or provide a message as well:

```
>>> response.status
'200 OK'
>>> response.status = '404 Not Found'
>>> response.status_code
404
>>> response.status_code = 400
>>> response.status
'400 BAD REQUEST'
```

As you can see attributes work in both directions. So you can set both `status` and `status_code` and the change will be reflected to the other.

Also common headers are exposed as attributes or with methods to set / retrieve them:

```
>>> response.content_length
12
>>> from datetime import datetime
>>> response.date = datetime(2009, 2, 20, 17, 42, 51)
>>> response.headers['Date']
'Fri, 20 Feb 2009 17:42:51 GMT'
```

Because etags can be weak or strong there are methods to set them:

```
>>> response.set_etag("12345-abcd")
>>> response.headers['etag']
'"12345-abcd"'
>>> response.get_etag()
('12345-abcd', False)
>>> response.set_etag("12345-abcd", weak=True)
>>> response.get_etag()
('12345-abcd', True)
```

Some headers are available as mutable structures. For example most of the *Content-* headers are sets of values:

```
>>> response.content_language.add('en-us')
>>> response.content_language.add('en')
>>> response.headers['Content-Language']
'en-us, en'
```

Also here this works in both directions:

```
>>> response.headers['Content-Language'] = 'de-AT, de'
>>> response.content_language
HeaderSet(['de-AT', 'de'])
```

Authentication headers can be set that way as well:

```
>>> response.www_authenticate.set_basic("My protected resource")
>>> response.headers['www-authenticate']
'Basic realm="My protected resource"'
```

Cookies can be set as well:

```
>>> response.set_cookie('name', 'value')
>>> response.headers['Set-Cookie']
'name=value; Path=/'
>>> response.set_cookie('name2', 'value2')
```

If headers appear multiple times you can use the `getlist()` method to get all values for a header:

```
>>> response.headers.getlist('Set-Cookie')
['name=value; Path=/', 'name2=value2; Path=/']
```

Finally if you have set all the conditional values, you can make the response conditional against a request. Which means that if the request can assure that it has the

information already, no data besides the headers is sent over the network which saves traffic. For that you should set at least an etag (which is used for comparison) and the date header and then call `make_conditional` with the request object.

The response is modified accordingly (status code changed, response body removed, entity headers removed etc.)

CHAPTER

SIX

PYTHON 3 NOTES

Since version 0.9, Werkzeug supports Python 3.3+ in addition to versions 2.6 and 2.7. Older Python 3 versions such as 3.2 or 3.1 are not supported.

This part of the documentation outlines special information required to use Werkzeug and WSGI on Python 3.

Warning: Python 3 support in Werkzeug is currently highly experimental. Please give feedback on it and help us improve it.

6.1 WSGI Environment

The WSGI environment on Python 3 works slightly different than it does on Python 2. For the most part Werkzeug hides the differences from you if you work on the higher level APIs. The main difference between Python 2 and Python 3 is that on Python 2 the WSGI environment contains bytes whereas the environment on Python 3 contains a range of differently encoded strings.

There are two different kinds of strings in the WSGI environ on Python 3:

- unicode strings restricted to latin1 values. These are the used for HTTP headers and a few other things.
- unicode strings carrying binary payload, roundtripped through latin1 values. This is usually referred as "WSGI encoding dance" throughout Werkzeug.

Werkzeug provides you with functionality to deal with these automatically so that you don't need to be aware of the inner workings. The following functions and classes should be used to read information out of the WSGI environment:

- *get_current_url()*
- *get_host()*
- *get_script_name()*
- *get_path_info()*
- *get_query_string()*

- *EnvironHeaders()*

Applications are strongly discouraged to create and modify a WSGI environment themselves on Python 3 unless they take care of the proper decoding step. All high level interfaces in Werkzeug will apply the correct encoding and decoding steps as necessary.

6.2 URLs

URLs in Werkzeug attempt to represent themselves as unicode strings on Python 3. All the parsing functions generally also provide functionality that allow operations on bytes. In some cases functions that deal with URLs allow passing in *None* as charset to change the return value to byte objects. Internally Werkzeug will now unify URIs and IRIs as much as possible.

6.3 Request Cleanup

Request objects on Python 3 and PyPy require explicit closing when file uploads are involved. This is required to properly close temporary file objects created by the multipart parser. For that purpose the `close()` method was introduced.

In addition to that request objects now also act as context managers that automatically close.

Part II

SERVING AND TESTING

The development server and testing support and management script utilities are covered here:

CHAPTER

SEVEN

SERVING WSGI APPLICATIONS

There are many ways to serve a WSGI application. While you're developing it, you usually don't want to have a full-blown webserver like Apache up and running, but instead a simple standalone one. Because of that Werkzeug comes with a builtin development server.

The easiest way is creating a small `start-myproject.py` file that runs the application using the builtin server:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from werkzeug.serving import run_simple
from myproject import make_app

app = make_app(...)
run_simple('localhost', 8080, app, use_reloader=True)
```

You can also pass it the *extra_files* keyword argument with a list of additional files (like configuration files) you want to observe.

`werkzeug.serving.run_simple`(*hostname*, *port*, *application*, *use_reloader=False*, *use_debugger=False*, *use_evalex=True*, *extra_files=None*, *reloader_interval=1*, *reloader_type='auto'*, *threaded=False*, *processes=1*, *request_handler=None*, *static_files=None*, *passthrough_errors=False*, *ssl_context=None*)

Start a WSGI application. Optional features include a reloader, multithreading and fork support.

This function has a command-line interface too:

```
python -m werkzeug.serving --help
```

New in version 0.5: *static_files* was added to simplify serving of static files as well as *passthrough_errors*.

New in version 0.6: support for SSL was added.

New in version 0.8: Added support for automatically loading a SSL context from certificate file and private key.

New in version 0.9: Added command-line interface.

New in version 0.10: Improved the reloader and added support for changing the backend through the *reloader_type* parameter. See *Reloader* for more information.

Parameters

- `hostname` – The host for the application. eg: `'localhost'`
- `port` – The port for the server. eg: `8080`
- `application` – the WSGI application to execute
- `use_reloader` – should the server automatically restart the python process if modules were changed?
- `use_debugger` – should the werkzeug debugging system be used?
- `use_evalex` – should the exception evaluation feature be enabled?
- `extra_files` – a list of files the reloader should watch additionally to the modules. For example configuration files.
- `reloader_interval` – the interval for the reloader in seconds.
- `reloader_type` – the type of reloader to use. The default is auto detection. Valid values are `'stat'` and `'watchdog'`. See *Reloader* for more information.
- `threaded` – should the process handle each request in a separate thread?
- `processes` – if greater than 1 then handle each request in a new process up to this maximum number of concurrent processes.
- `request_handler` – optional parameter that can be used to replace the default one. You can use this to replace it with a different `BaseHTTPRequestHandler` subclass.
- `static_files` – a dict of paths for static files. This works exactly like `SharedDataMiddleware`, it's actually just wrapping the application in that middleware before serving.
- `passthrough_errors` – set this to *True* to disable the error catching. This means that the server will die on errors but it can be useful to hook debuggers in (pdb etc.)
- `ssl_context` – an SSL context for the connection. Either an `ssl.SSLContext`, a tuple in the form `(cert_file, pkey_file)`, the string `'adhoc'` if the server should automatically create one, or `None` to disable SSL (which is the default).

`werkzeug.serving.is_running_from_reloader()`

Checks if the application is running from within the Werkzeug reloader subprocess.

New in version 0.10.

`werkzeug.serving.make_ssl_devcert`(*base_path*, *host=None*, *cn=None*)

Creates an SSL key for development. This should be used instead of the `'adhoc'` key which generates a new cert on each server start. It accepts a path for where it should store the key and cert and either a host or CN. If a host is given it will use the CN `*.host/CN=host`.

For more information see *run_simple()*.

New in version 0.9.

Parameters

- `base_path` – the path to the certificate and key. The extension `.crt` is added for the certificate, `.key` is added for the key.
- `host` – the name of the host. This can be used as an alternative for the *cn*.
- `cn` – the *CN* to use.

Information

The development server is not intended to be used on production systems. It was designed especially for development purposes and performs poorly under high load. For deployment setups have a look at the *Application Deployment* pages.

7.1 Reloader

Changed in version 0.10.

The Werkzeug reloader constantly monitors modules and paths of your web application, and restarts the server if any of the observed files change.

Since version 0.10, there are two backends the reloader supports: `stat` and `watchdog`.

- The default `stat` backend simply checks the `mtime` of all files in a regular interval. This is sufficient for most cases, however, it is known to drain a laptop's battery.
- The `watchdog` backend uses filesystem events, and is much faster than `stat`. It requires the watchdog module to be installed.

If `watchdog` is installed and available it will automatically be used instead of the builtin `stat` reloader.

To switch between the backends you can use the *reloader_type* parameter of the *run_simple()* function. `'stat'` sets it to the default stat based polling and `'watchdog'` forces it to the watchdog backend.

Note: Some edge cases, like modules that failed to import correctly, are not handled by the stat reloader for performance reasons. The watchdog reloader monitors such files too.

7.2 Virtual Hosts

Many web applications utilize multiple subdomains. This can be a bit tricky to simulate locally. Fortunately there is the hosts file that can be used to assign the local computer multiple names.

This allows you to call your local computer *yourapplication.local* and *api.yourapplication.local* (or anything else) in addition to *localhost*.

You can find the hosts file on the following location:

Windows	`%SystemRoot%\system32\drivers\etc\hosts`
Linux / OS X	`/etc/hosts`

You can open the file with your favorite text editor and add a new name after *localhost*:

```
127.0.0.1       localhost yourapplication.local api.yourapplication.local
```

Save the changes and after a while you should be able to access the development server on these host names as well. You can use the *URL Routing* system to dispatch between different hosts or parse `request.host` yourself.

7.3 Shutting Down The Server

New in version 0.7.

Starting with Werkzeug 0.7 the development server provides a way to shut down the server after a request. This currently only works with Python 2.6 and later and will only work with the development server. To initiate the shutdown you have to call a function named `'werkzeug.server.shutdown'` in the WSGI environment:

```
def shutdown_server(environ):
    if not 'werkzeug.server.shutdown' in environ:
        raise RuntimeError('Not running the development server')
    environ['werkzeug.server.shutdown']()
```

7.4 Troubleshooting

On operating systems that support ipv6 and have it configured such as modern Linux systems, OS X 10.4 or higher as well as Windows Vista some browsers can be painfully slow if accessing your local server. The reason for this is that sometimes "localhost" is configured to be available on both ipv4 and ipv6 socktes and some browsers will try to access ipv6 first and then ivp4.

At the current time the integrated webserver does not support ipv6 and ipv4 at the same time and for better portability ipv4 is the default.

If you notice that the web browser takes ages to load the page there are two ways around this issue. If you don't need ipv6 support you can disable the ipv6 entry in the hosts file by removing this line:

```
::1             localhost
```

Alternatively you can also disable ipv6 support in your browser. For example if Firefox shows this behavior you can disable it by going to `about:config` and disabling the *network.dns.disableIPv6* key. This however is not recommended as of Werkzeug 0.6.1!

Starting with Werkzeug 0.6.1, the server will now switch between ipv4 and ipv6 based on your operating system's configuration. This means if that you disabled ipv6 support in your browser but your operating system is preferring ipv6, you will be unable to connect to your server. In that situation, you can either remove the localhost entry for `::1` or explicitly bind the hostname to an ipv4 address (*127.0.0.1*)

7.5 SSL

New in version 0.6.

The builtin server supports SSL for testing purposes. If an SSL context is provided it will be used. That means a server can either run in HTTP or HTTPS mode, but not both.

7.5.1 Quickstart

The easiest way to do SSL based development with Werkzeug is by using it to generate an SSL certificate and private key and storing that somewhere and to then put it there. For the certificate you need to provide the name of your server on generation or a *CN*.

1. Generate an SSL key and store it somewhere:

```
>>> from werkzeug.serving import make_ssl_devcert
>>> make_ssl_devcert('/path/to/the/key', host='localhost')
('/path/to/the/key.crt', '/path/to/the/key.key')
```

2. Now this tuple can be passed as `ssl_context` to the *`run_simple()`* method:

```
run_simple('localhost', 4000, application,
           ssl_context=('/path/to/the/key.crt',
                        '/path/to/the/key.key'))
```

You will have to acknowledge the certificate in your browser once then.

7.5.2 Loading Contexts by Hand

In Python 2.7.9 and 3+ you also have the option to use a `ssl.SSLContext` object instead of a simple tuple. This way you have better control over the SSL behavior of Werkzeug's builtin server:

```
import ssl
ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
ctx.load_cert_chain('ssl.cert', 'ssl.key')
run_simple('localhost', 4000, application, ssl_context=ctx)
```

7.5.3 Generating Certificates

A key and certificate can be created in advance using the openssl tool instead of the *`make_ssl_devcert()`*. This requires that you have the *openssl* command installed on your system:

```
$ openssl genrsa 1024 > ssl.key
$ openssl req -new -x509 -nodes -sha1 -days 365 -key ssl.key > ssl.cert
```

7.5.4 Adhoc Certificates

The easiest way to enable SSL is to start the server in adhoc-mode. In that case Werkzeug will generate an SSL certificate for you:

```
run_simple('localhost', 4000, application,
           ssl_context='adhoc')
```

The downside of this of course is that you will have to acknowledge the certificate each time the server is reloaded. Adhoc certificates are discouraged because modern browsers do a bad job at supporting them for security reasons.

This feature requires the pyOpenSSL library to be installed.

CHAPTER

EIGHT

TEST UTILITIES

Quite often you want to unittest your application or just check the output from an interactive python session. In theory that is pretty simple because you can fake a WSGI environment and call the application with a dummy *start_response* and iterate over the application iterator but there are argumentably better ways to interact with an application.

8.1 Diving In

Werkzeug provides a *Client* object which you can pass a WSGI application (and optionally a response wrapper) which you can use to send virtual requests to the application.

A response wrapper is a callable that takes three arguments: the application iterator, the status and finally a list of headers. The default response wrapper returns a tuple. Because response objects have the same signature, you can use them as response wrapper, ideally by subclassing them and hooking in test functionality.

```
>>> from werkzeug.test import Client
>>> from werkzeug.testapp import test_app
>>> from werkzeug.wrappers import BaseResponse
>>> c = Client(test_app, BaseResponse)
>>> resp = c.get('/')
>>> resp.status_code
200
>>> resp.headers
Headers([('Content-Type', 'text/html; charset=utf-8'), ('Content-Length', '8339')])
>>> resp.data.splitlines()[0]
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"'
```

Or without a wrapper defined:

```
>>> c = Client(test_app)
>>> app_iter, status, headers = c.get('/')
>>> status
'200 OK'
>>> headers
[('Content-Type', 'text/html; charset=utf-8'), ('Content-Length', '8339')]
```

```
>>> ''.join(app_iter).splitlines()[0]
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"'
```

8.2 Environment Building

New in version 0.5.

The easiest way to interactively test applications is using the *EnvironBuilder*. It can create both standard WSGI environments and request objects.

The following example creates a WSGI environment with one uploaded file and a form field:

```
>>> from werkzeug.test import EnvironBuilder
>>> from StringIO import StringIO
>>> builder = EnvironBuilder(method='POST', data={'foo': 'this is some text',
...      'file': (StringIO('my file contents'), 'test.txt')})
>>> env = builder.get_environ()
```

The resulting environment is a regular WSGI environment that can be used for further processing:

```
>>> from werkzeug.wrappers import Request
>>> req = Request(env)
>>> req.form['foo']
u'this is some text'
>>> req.files['file']
<FileStorage: u'test.txt' ('text/plain')>
>>> req.files['file'].read()
'my file contents'
```

The *EnvironBuilder* figures out the content type automatically if you pass a dict to the constructor as *data*. If you provide a string or an input stream you have to do that yourself.

By default it will try to use `application/x-www-form-urlencoded` and only use `multipart/form-data` if files are uploaded:

```
>>> builder = EnvironBuilder(method='POST', data={'foo': 'bar'})
>>> builder.content_type
'application/x-www-form-urlencoded'
>>> builder.files['foo'] = StringIO('contents')
>>> builder.content_type
'multipart/form-data'
```

If a string is provided as data (or an input stream) you have to specify the content type yourself:

```
>>> builder = EnvironBuilder(method='POST', data='{"json": "this is"}')
>>> builder.content_type
>>> builder.content_type = 'application/json'
```

8.3 Testing API

class `werkzeug.test.EnvironBuilder`(*path='/'*, *base_url=None*, *query_string=None*, *method='GET'*, *input_stream=None*, *content_type=None*, *content_length=None*, *errors_stream=None*, *multithread=False*, *multiprocess=False*, *run_once=False*, *headers=None*, *data=None*, *environ_base=None*, *environ_overrides=None*, *charset='utf-8'*)

This class can be used to conveniently create a WSGI environment for testing purposes. It can be used to quickly create WSGI environments or request objects from arbitrary data.

The signature of this class is also used in some other places as of Werkzeug 0.5 (*create_environ()*, `BaseResponse.from_values()`, *Client.open()*). Because of this most of the functionality is available through the constructor alone.

Files and regular form data can be manipulated independently of each other with the `form` and `files` attributes, but are passed with the same argument to the constructor: *data*.

data can be any of these values:

- a *str*: If it's a string it is converted into a *input_stream*, the *content_length* is set and you have to provide a *content_type*.
- a *dict*: If it's a dict the keys have to be strings and the values any of the following objects:
  - a `file`-like object. These are converted into `FileStorage` objects automatically.
  - a tuple. The `add_file()` method is called with the tuple items as positional arguments.

New in version 0.6: *path* and *base_url* can now be unicode strings that are encoded using the `iri_to_uri()` function.

Parameters

- `path` – the path of the request. In the WSGI environment this will end up as *PATH_INFO*. If the *query_string* is not defined and there is a question mark in the *path* everything after it is used as query string.
- `base_url` – the base URL is a URL that is used to extract the WSGI URL scheme, host (server name + server port) and the script root (*SCRIPT_NAME*).

- `query_string` – an optional string or dict with URL parameters.
- `method` – the HTTP method to use, defaults to *GET*.
- `input_stream` – an optional input stream. Do not specify this and *data*. As soon as an input stream is set you can't modify *args* and `files` unless you set the *input_stream* to *None* again.
- `content_type` – The content type for the request. As of 0.5 you don't have to provide this when specifying files and form data via *data*.
- `content_length` – The content length for the request. You don't have to specify this when providing data via *data*.
- `errors_stream` – an optional error stream that is used for *wsgi.errors*. Defaults to `stderr`.
- `multithread` – controls *wsgi.multithread*. Defaults to *False*.
- `multiprocess` – controls *wsgi.multiprocess*. Defaults to *False*.
- `run_once` – controls *wsgi.run_once*. Defaults to *False*.
- `headers` – an optional list or `Headers` object of headers.
- `data` – a string or dict of form data. See explanation above.
- `environ_base` – an optional dict of environment defaults.
- `environ_overrides` – an optional dict of environment overrides.
- `charset` – the charset used to encode unicode data.

`path`
: The path of the application. (aka *PATH_INFO*)

`charset`
: The charset used to encode unicode data.

`headers`
: A `Headers` object with the request headers.

`errors_stream`
: The error stream used for the *wsgi.errors* stream.

`multithread`
: The value of *wsgi.multithread*

`multiprocess`
: The value of *wsgi.multiprocess*

`environ_base`
: The dict used as base for the newly create environ.

`environ_overrides`
: A dict with values that are used to override the generated environ.

`input_stream`
: The optional input stream. This and `form` / `files` is mutually exclusive. Also do not provide this stream if the request method is not *POST* / *PUT* or something comparable.

`args`
: The URL arguments as `MultiDict`.

`base_url`
: The base URL is a URL that is used to extract the WSGI URL scheme, host (server name + server port) and the script root (*SCRIPT_NAME*).

`close()`
: Closes all files. If you put real `file` objects into the `files` dict you can call this method to automatically close them all in one go.

`content_length`
: The content length as integer. Reflected from and to the *`headers`*. Do not set if you set `files` or `form` for auto detection.

`content_type`
: The content type for the request. Reflected from and to the *`headers`*. Do not set if you set `files` or `form` for auto detection.

`get_environ()`
: Return the built environ.

`get_request(`*cls=None*`)`
: Returns a request with the data. If the request class is not specified *`request_class`* is used.

  Parameters `cls` – The request wrapper to use.

`input_stream`
: An optional input stream. If you set this it will clear `form` and `files`.

`query_string`
: The query string. If you set this to a string *`args`* will no longer be available.

`request_class`
: the default request class for *`get_request()`*

  alias of `BaseRequest`

`server_name`
: The server name (read-only, use `host` to set)

`server_port`
: The server port as integer (read-only, use `host` to set)

`server_protocol` **= 'HTTP/1.1'**
: the server protocol to use. defaults to HTTP/1.1

`wsgi_version` **= (1, 0)**
the wsgi version to use. defaults to (1, 0)

class `werkzeug.test.Client`(*application*, *response_wrapper=None*, *use_cookies=True*, *allow_subdomain_redirects=False*)
This class allows to send requests to a wrapped application.

The response wrapper can be a class or factory function that takes three arguments: app_iter, status and headers. The default response wrapper just returns a tuple.

Example:

```
class ClientResponse(BaseResponse):
    ...

client = Client(MyApplication(), response_wrapper=ClientResponse)
```

The use_cookies parameter indicates whether cookies should be stored and sent for subsequent requests. This is True by default, but passing False will disable this behaviour.

If you want to request some subdomain of your application you may set *allow_subdomain_redirects* to *True* as if not no external redirects are allowed.

New in version 0.5: *use_cookies* is new in this version. Older versions did not provide builtin cookie support.

`open`(**args*, ***kwargs*)
Takes the same arguments as the *EnvironBuilder* class with some additions: You can provide a *EnvironBuilder* or a WSGI environment as only argument instead of the *EnvironBuilder* arguments and two optional keyword arguments (*as_tuple*, *buffered*) that change the type of the return value or the way the application is executed.

Changed in version 0.5: If a dict is provided as file in the dict for the *data* parameter the content type has to be called *content_type* now instead of *mimetype*. This change was made for consistency with `werkzeug.FileWrapper`.

The *follow_redirects* parameter was added to *open()*.

Additional parameters:

Parameters

- `as_tuple` – Returns a tuple in the form `(environ, result)`
- `buffered` – Set this to True to buffer the application run. This will automatically close the application for you as well.
- `follow_redirects` – Set this to True if the *Client* should follow HTTP redirects.

Shortcut methods are available for many HTTP methods:

`get(*args, **kw)`
Like open but method is enforced to GET.

`patch(*args, **kw)`
Like open but method is enforced to PATCH.

`post(*args, **kw)`
Like open but method is enforced to POST.

`head(*args, **kw)`
Like open but method is enforced to HEAD.

`put(*args, **kw)`
Like open but method is enforced to PUT.

`delete(*args, **kw)`
Like open but method is enforced to DELETE.

`options(*args, **kw)`
Like open but method is enforced to OPTIONS.

`trace(*args, **kw)`
Like open but method is enforced to TRACE.

`werkzeug.test.create_environ([options])`
Create a new WSGI environ dict based on the values passed. The first parameter should be the path of the request which defaults to '/'. The second one can either be an absolute path (in that case the host is localhost:80) or a full path to the request with scheme, netloc port and the path to the script.

This accepts the same arguments as the *EnvironBuilder* constructor.

Changed in version 0.5: This function is now a thin wrapper over *EnvironBuilder* which was added in 0.5. The *headers*, *environ_base*, *environ_overrides* and *charset* parameters were added.

`werkzeug.test.run_wsgi_app(app, environ, buffered=False)`
Return a tuple in the form (app_iter, status, headers) of the application output. This works best if you pass it an application that returns an iterator all the time.

Sometimes applications may use the *write()* callable returned by the *start_response* function. This tries to resolve such edge cases automatically. But if you don't get the expected output you should set *buffered* to *True* which enforces buffering.

If passed an invalid WSGI application the behavior of this function is undefined. Never pass non-conforming WSGI applications to this function.

Parameters

- `app` – the application to execute.
- `buffered` – set to *True* to enforce buffering.

Returns tuple in the form `(app_iter, status, headers)`

CHAPTER

NINE

DEBUGGING APPLICATIONS

Depending on the WSGI gateway/server, exceptions are handled differently. But most of the time, exceptions go to stderr or the error log.

Since this is not the best debugging environment, Werkzeug provides a WSGI middleware that renders nice debugging tracebacks, optionally with an AJAX based debugger (which allows to execute code in the context of the traceback's frames).

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

9.1 Enabling the Debugger

You can enable the debugger by wrapping the application in a *DebuggedApplication* middleware. Additionally there are parameters to the `run_simple()` function to enable it because this is a common task during development.

class `werkzeug.debug.DebuggedApplication`(*app*, *evalex=False*, *request_key='werkzeug.request'*, *console_path='/console'*, *console_init_func=None*, *show_hidden_frames=False*, *lodgeit_url=None*)

Enables debugging support for a given application:

```
from werkzeug.debug import DebuggedApplication
from myapp import app
app = DebuggedApplication(app, evalex=True)
```

The *evalex* keyword argument allows evaluating expressions in a traceback's frame context.

New in version 0.9: The *lodgeit_url* parameter was deprecated.

Parameters

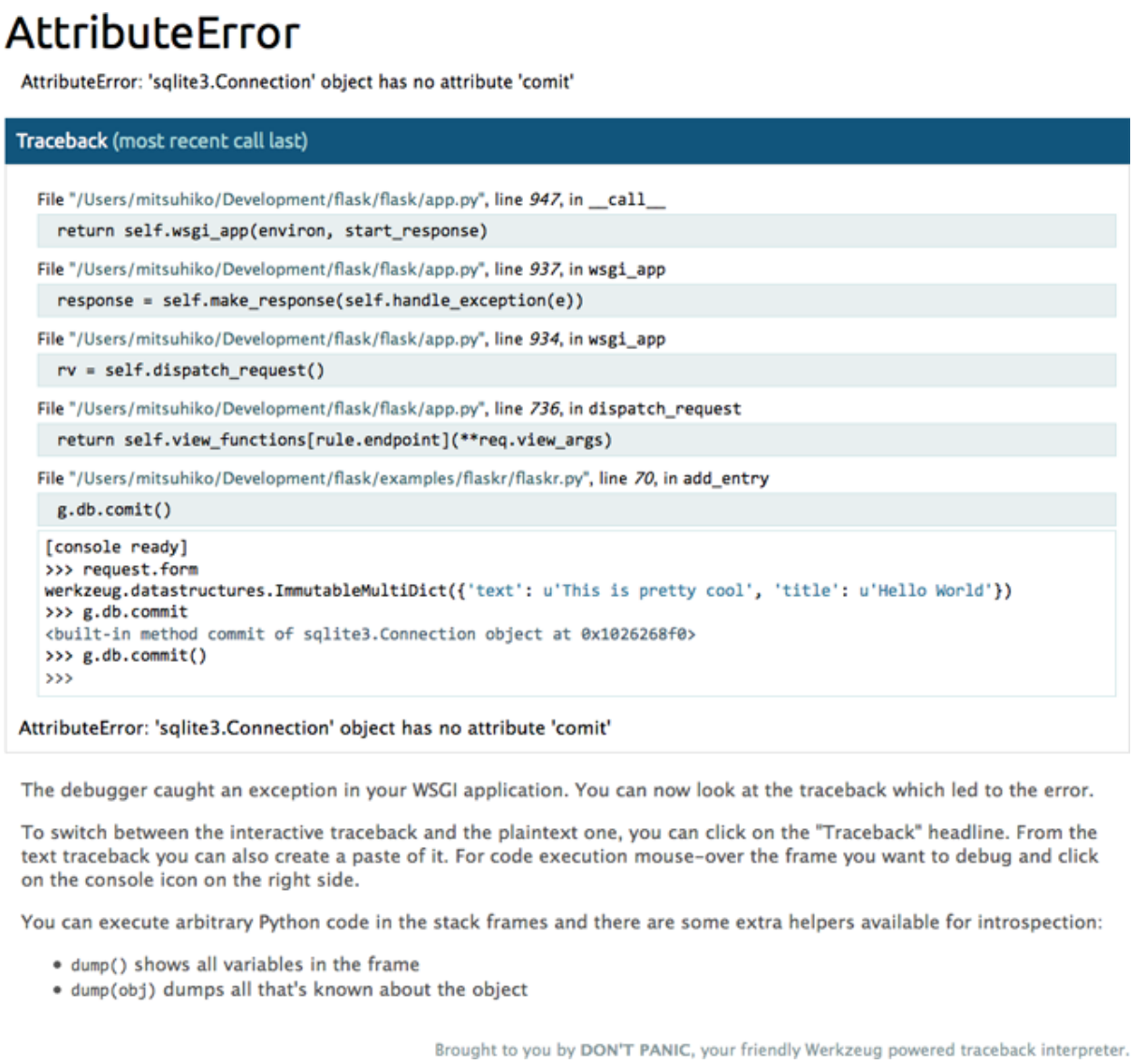
- `app` – the WSGI application to run debugged.

- `evalex` – enable exception evaluation feature (interactive debugging). This requires a non-forking server.
- `request_key` – The key that points to the request object in ths environment. This parameter is ignored in current versions.
- `console_path` – the URL for a general purpose console.
- `console_init_func` – the function that is executed before starting the general purpose console. The return value is used as initial namespace.
- `show_hidden_frames` – by default hidden traceback frames are skipped. You can show them by setting this parameter to *True*.

9.2 Using the Debugger

Once enabled and an error happens during a request you will see a detailed traceback instead of a general "internal server error". If you have the *evalex* feature enabled you can also get a traceback for every frame in the traceback by clicking on the console icon.

Once clicked a console opens where you can execute Python code in:



Inside the interactive consoles you can execute any kind of Python code. Unlike regular Python consoles the output of the object reprs is colored and stripped to a reasonable size by default. If the output is longer than what the console decides to display a small plus sign is added to the repr and a click will expand the repr.

To display all variables that are defined in the current frame you can use the *dump()* function. You can call it without arguments to get a detailed list of all variables and their values, or with an object as argument to get a detailed list of all the attributes it has.

9.3 Pasting Errors

If you click on the *Traceback* title the traceback switches over to a text based one. The text based one can be pasted to paste.pocoo.org with one click.

Part III
REFERENCE

CHAPTER

TEN

REQUEST / RESPONSE OBJECTS

The request and response objects wrap the WSGI environment or the return value from a WSGI application so that it is another WSGI application (wraps a whole application).

10.1 How they Work

Your WSGI application is always passed two arguments. The WSGI "environment" and the WSGI *start_response* function that is used to start the response phase. The *Request* class wraps the *environ* for easier access to request variables (form data, request headers etc.).

The *Response* on the other hand is a standard WSGI application that you can create. The simple hello world in Werkzeug looks like this:

```
from werkzeug.wrappers import Response
application = Response('Hello World!')
```

To make it more useful you can replace it with a function and do some processing:

```
from werkzeug.wrappers import Request, Response

def application(environ, start_response):
    request = Request(environ)
    response = Response("Hello %s!" % request.args.get('name', 'World!'))
    return response(environ, start_response)
```

Because this is a very common task the *Request* object provides a helper for that. The above code can be rewritten like this:

```
from werkzeug.wrappers import Request, Response

@Request.application
def application(request):
    return Response("Hello %s!" % request.args.get('name', 'World!'))
```

The *application* is still a valid WSGI application that accepts the environment and *start_response* callable.

10.2 Mutability and Reusability of Wrappers

The implementation of the Werkzeug request and response objects are trying to guard you from common pitfalls by disallowing certain things as much as possible. This serves two purposes: high performance and avoiding of pitfalls.

For the request object the following rules apply:

1. The request object is immutable. Modifications are not supported by default, you may however replace the immutable attributes with mutable attributes if you need to modify it.
2. The request object may be shared in the same thread, but is not thread safe itself. If you need to access it from multiple threads, use locks around calls.
3. It's not possible to pickle the request object.

For the response object the following rules apply:

1. The response object is mutable
2. The response object can be pickled or copied after *freeze()* was called.
3. Since Werkzeug 0.6 it's safe to use the same response object for multiple WSGI responses.
4. It's possible to create copies using *copy.deepcopy*.

10.3 Base Wrappers

These objects implement a common set of operations. They are missing fancy addon functionality like user agent parsing or etag handling. These features are available by mixing in various mixin classes or using `Request` and `Response`.

class `werkzeug.wrappers.BaseRequest`(*environ*, *populate_request=True*, *shallow=False*)

Very basic request object. This does not implement advanced stuff like entity tag parsing or cache controls. The request object is created with the WSGI environment as first argument and will add itself to the WSGI environment as `'werkzeug.request'` unless it's created with *populate_request* set to False.

There are a couple of mixins available that add additional functionality to the request object, there is also a class called *Request* which subclasses *BaseRequest* and all the important mixins.

It's a good idea to create a custom subclass of the `BaseRequest` and add missing functionality either via mixins or direct implementation. Here an example for such subclasses:

```
from werkzeug.wrappers import BaseRequest, ETagRequestMixin
```

```
class Request(BaseRequest, ETagRequestMixin):
    pass
```

Request objects are **read only**. As of 0.5 modifications are not allowed in any place. Unlike the lower level parsing functions the request object will use immutable objects everywhere possible.

Per default the request object will assume all the text data is *utf-8* encoded. Please refer to the unicode chapter for more details about customizing the behavior.

Per default the request object will be added to the WSGI environment as *werkzeug.request* to support the debugging system. If you don't want that, set *populate_request* to *False*.

If *shallow* is *True* the environment is initialized as shallow object around the environ. Every operation that would modify the environ in any way (such as consuming form data) raises an exception unless the *shallow* attribute is explicitly set to *False*. This is useful for middlewares where you don't want to consume the form data by accident. A shallow request is not populated to the WSGI environment.

Changed in version 0.5: read-only mode was enforced by using immutables classes for all data.

`environ`
: The WSGI environment that the request object uses for data retrival.

`shallow`
: *True* if this request object is shallow (does not modify *environ*), *False* otherwise.

`_get_file_stream`(*total_content_length*, *content_type*, *filename=None*, *content_length=None*)
: Called to get a stream for the file upload.

  This must provide a file-like class with *read()*, *readline()* and *seek()* methods that is both writeable and readable.

  The default implementation returns a temporary file if the total content length is higher than 500KB. Because many browsers do not provide a content length for the files only the total content length matters.

  Parameters

  - `total_content_length` – the total content length of all the data in the request combined. This value is guaranteed to be there.
  - `content_type` – the mimetype of the uploaded file.
  - `filename` – the filename of the uploaded file. May be *None*.
  - `content_length` – the length of this file. This value is usually not provided because webbrowsers do not provide this value.

`access_route`
: If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.

classmethod `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!')
```

Parameters `f` – the WSGI callable to decorate

Returns a new WSGI callable

`args`
: The parsed URL 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.

`base_url`
: Like *url* but without the querystring See also: *trusted_hosts*.

`charset` = **'utf-8'**
: the charset for the request, defaults to utf-8

`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 with will automatically close it.

  New in version 0.9.

`cookies`
: Read only access to the retrieved cookie values as dictionary.

`dict_storage_class`
: the type to be used for dict values from the incoming WSGI environment. By default an *ImmutableTypeConversionDict* is used (for example for *cookies*).

  New in version 0.6.

  alias of `ImmutableTypeConversionDict`

`disable_data_descriptor` = **False**
: Indicates whether the data descriptor should be allowed to read and buffer up the input stream. By default it's enabled.

  New in version 0.9.

`encoding_errors` = **'replace'**
: the error handling procedure for errors, defaults to 'replace'

`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.

  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.

`form_data_parser_class`
: The form data parser that shoud be used. Can be replaced to customize the form date parsing.

  alias of `FormDataParser`

classmethod `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.

`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*.

`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 exeuction 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.

`list_storage_class`
: the type to be used for list values from the incoming WSGI environment. By default an *ImmutableList* is used (for example for `access_list`).

  New in version 0.6.

  alias of `ImmutableList`

`make_form_data_parser()`
: Creates the form data parser. Instanciates the *form_data_parser_class*

with some parameters.

New in version 0.8.

`max_content_length` = **None**
: the maximum content length. This is forwarded to the form data parsing function (`parse_form_data()`). When set and the *`form`* or *`files`* attribute is accessed and the parsing fails because more than the specified value is transmitted a *`RequestEntityTooLarge`* exception is raised.

Have a look at *Dealing with Request Data* for more details.

New in version 0.5.

`max_form_memory_size` = **None**
: the maximum form field size. This is forwarded to the form data parsing function (`parse_form_data()`). When set and the *`form`* or *`files`* attribute is accessed and the data in memory for post data is longer than the specified value a *`RequestEntityTooLarge`* exception is raised.

Have a look at *Dealing with Request Data* for more details.

New in version 0.5.

`method`
: The transmission method. (For example `'GET'` or `'POST'`).

`parameter_storage_class`
: the class to use for *args* and *form*. The default is an *`ImmutableMultiDict`* which supports multiple values per key. alternatively it makes sense to use an *`ImmutableOrderedMultiDict`* which preserves order or a *`ImmutableDict`* which is the fastest but only remembers the last key. It is also possible to use mutable structures, but this is not recommended.

New in version 0.6.

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.

`query_string`
: The URL parameters as raw bytestring.

`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.

`scheme`
: URL scheme (http or https).

New in version 0.7.

`script_root`
The root path of the script without the trailing slash.

`stream`
The stream to read incoming data from. 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.

`trusted_hosts` **= None**
Optionally a list of hosts that is trusted by this request. By default all hosts are trusted which means that whatever the client sends the host is will be accepted.

This is the recommended setup as a webserver should manually be set up to only route correct hosts to the application, and remove the *X-Forwarded-Host* header if it is not being used (see *werkzeug.wsgi.get_host()*).

New in version 0.9.

`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*.

`values`
Combined multi dict for *args* and *form*.

`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.

class `werkzeug.wrappers.BaseResponse`(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)
Base response class. The most important fact about a response object is that it's a regular WSGI application. It's initialized with a couple of response parameters (headers, body, status code etc.) and will start a valid WSGI response when called with the environ and start response callable.

Because it's a WSGI application itself processing usually ends before the actual response is sent to the server. This helps debugging systems because they can catch all the exceptions before responses are started.

Here a small example WSGI application that takes advantage of the response objects:

```
from werkzeug.wrappers import BaseResponse as Response

def index():
    return Response('Index page')

def application(environ, start_response):
    path = environ.get('PATH_INFO') or '/'
    if path == '/':
        response = index()
    else:
        response = Response('Not Found', status=404)
    return response(environ, start_response)
```

Like *BaseRequest* which object is lacking a lot of functionality implemented in mixins. This gives you a better control about the actual API of your response objects, so you can create subclasses and add custom functionality. A full featured response object is available as *Response* which implements a couple of useful mixins.

To enforce a new type of already existing responses you can use the *force_type()* method. This is useful if you're working with different subclasses of response objects and you want to post process them with a know interface.

Per default the request object will assume all the text data is *utf-8* encoded. Please refer to the unicode chapter for more details about customizing the behavior.

Response can be any kind of iterable or string. If it's a string it's considered being an iterable with one item which is the string passed. Headers can be a list of tuples or a *Headers* object.

Special note for *mimetype* and *content_type*: For most mime types *mimetype* and *content_type* work the same, the difference affects only 'text' mimetypes. If the mimetype passed with *mimetype* is a mimetype starting with *text/*, the charset parameter of the response object is appended to it. In contrast the *content_type* parameter is always added as header unmodified.

Changed in version 0.5: the *direct_passthrough* parameter was added.

Parameters

- `response` – a string or response iterable.
- `status` – a string with a status or an integer with the status code.
- `headers` – a list of headers or a *Headers* object.
- `mimetype` – the mimetype for the request. See notice above.

- `content_type` – the content type for the request. See notice above.
- `direct_passthrough` – if set to *True* *iter_encoded()* is not called before iteration which makes it possible to pass special iterators though unchanged (see `wrap_file()` for more details.)

`response`
: The application iterator. If constructed from a string this will be a list, otherwise the object provided as application iterator. (The first argument passed to *BaseResponse*)

`headers`
: A `Headers` object representing the response headers.

`status_code`
: The response status as integer.

`direct_passthrough`
: If `direct_passthrough=True` was passed to the response object or if this attribute was set to *True* before using the response object as WSGI application, the wrapped iterator is returned unchanged. This makes it possible to pass a special *wsgi.file_wrapper* to the response object. See `wrap_file()` for more details.

`__call__`(*environ*, *start_response*)
: Process this response as WSGI application.

  Parameters

  - `environ` – the WSGI environment.
  - `start_response` – the response callable provided by the WSGI server.

  Returns an application iterator

`_ensure_sequence`(*mutable=False*)
: This method can be called by methods that need a sequence. If *mutable* is true, it will also ensure that the response sequence is a standard Python list.

  New in version 0.6.

`autocorrect_location_header` = **True**
: Should this response object correct the location header to be RFC conformant? This is true by default.

  New in version 0.8.

`automatically_set_content_length` = **True**
: Should this response object automatically set the content-length header if possible? This is true by default.

  New in version 0.8.

`calculate_content_length()`
Returns the content length if available or *None* otherwise.

`call_on_close(`*func*`)`
Adds a function to the internal list of functions that should be called as part of closing down the response. Since 0.7 this function also returns the function that was passed so that this can be used as a decorator.

New in version 0.6.

`charset` **= 'utf-8'**
the charset of the response.

`close()`
Close the wrapped response if possible. You can also use the object in a with statement which will automatically close it.

New in version 0.9: Can now be used in a with statement.

`data`
A descriptor that calls *get_data()* and *set_data()*. This should not be used and will eventually get deprecated.

`default_mimetype` **= 'text/plain'**
the default mimetype if none is provided.

`default_status` **= 200**
the default status if none is provided.

`delete_cookie(`*key*, *path='/'*, *domain=None*`)`
Delete a cookie. Fails silently if key doesn't exist.

Parameters

- `key` – the key (name) of the cookie to be deleted.
- `path` – if the cookie that should be deleted was limited to a path, the path has to be defined here.
- `domain` – if the cookie that should be deleted was limited to a domain, that domain has to be defined here.

classmethod `force_type(`*response*, *environ=None*`)`
Enforce that the WSGI response is a response object of the current type. Werkzeug will use the *BaseResponse* internally in many situations like the exceptions. If you call `get_response()` on an exception you will get back a regular *BaseResponse* object, even if you are using a custom subclass.

This method can enforce a given response type, and it will also convert arbitrary WSGI callables into response objects if an environ is provided:

```
# convert a Werkzeug response object into an instance of the
# MyResponseClass subclass.
response = MyResponseClass.force_type(response)
```

```
# convert any WSGI application into a response object
response = MyResponseClass.force_type(response, environ)
```

This is especially useful if you want to post-process responses in the main dispatcher and use functionality provided by your subclass.

Keep in mind that this will modify response objects in place if possible!

Parameters

- `response` – a response object or wsgi application.
- `environ` – a WSGI environment object.

Returns a response object.

`freeze()`
Call this method if you want to make your response object ready for being pickled. This buffers the generator if there is one. It will also set the *Content-Length* header to the length of the body.

Changed in version 0.6: The *Content-Length* header is now set.

classmethod `from_app(`*app*, *environ*, *buffered=False*`)`
Create a new response object from an application output. This works best if you pass it an application that returns a generator all the time. Sometimes applications may use the *write()* callable returned by the *start_response* function. This tries to resolve such edge cases automatically. But if you don't get the expected output you should set *buffered* to *True* which enforces buffering.

Parameters

- `app` – the WSGI application to execute.
- `environ` – the WSGI environment to execute against.
- `buffered` – set to *True* to enforce buffering.

Returns a response object.

`get_app_iter(`*environ*`)`
Returns the application iterator for the given environ. Depending on the request method and the current status code the return value might be an empty response rather than the one from the response.

If the request method is *HEAD* or the status code is in a range where the HTTP specification requires an empty response, an empty iterable is returned.

New in version 0.6.

Parameters `environ` – the WSGI environment of the request.

Returns a response iterable.

`get_data(`*as_text=False*`)`
The string representation of the request body. Whenever you call this prop-

erty the request iterable is encoded and flattened. This can lead to unwanted behavior if you stream big data.

This behavior can be disabled by setting *implicit_sequence_conversion* to *False*.

If *as_text* is set to *True* the return value will be a decoded unicode string.

New in version 0.9.

`get_wsgi_headers`(*environ*)
This is automatically called right before the response is started and returns headers modified for the given environment. It returns a copy of the headers from the response with some modifications applied if necessary.

For example the location header (if present) is joined with the root URL of the environment. Also the content length is automatically set to zero here for certain status codes.

Changed in version 0.6: Previously that function was called *fix_headers* and modified the response object in place. Also since 0.6, IRIs in location and content-location headers are handled properly.

Also starting with 0.6, Werkzeug will attempt to set the content length if it is able to figure it out on its own. This is the case if all the strings in the response iterable are already encoded and the iterable is buffered.

Parameters `environ` – the WSGI environment of the request.

Returns returns a new *Headers* object.

`get_wsgi_response`(*environ*)
Returns the final WSGI response as tuple. The first item in the tuple is the application iterator, the second the status and the third the list of headers. The response returned is created specially for the given environment. For example if the request method in the WSGI environment is `'HEAD'` the response will be empty and only the headers and status code will be present.

New in version 0.6.

Parameters `environ` – the WSGI environment of the request.

Returns an `(app_iter, status, headers)` tuple.

`implicit_sequence_conversion` = **True**
if set to *False* accessing properties on the response object will not try to consume the response iterator and convert it into a list.

New in version 0.6.2: That attribute was previously called *implicit_seqence_conversion*. (Notice the typo). If you did use this feature, you have to adapt your code to the name change.

`is_sequence`
If the iterator is buffered, this property will be *True*. A response object will consider an iterator to be buffered if the response attribute is a list or tuple.

New in version 0.6.

`is_streamed`
: If the response is streamed (the response is not an iterable with a length information) this property is *True*. In this case streamed means that there is no information about the number of iterations. This is usually *True* if a generator is passed to the response object.

  This is useful for checking before applying some sort of post filtering that should not take place for streamed responses.

`iter_encoded()`
: Iter the response encoded with the encoding of the response. If the response object is invoked as WSGI application the return value of this method is used as application iterator unless `direct_passthrough` was activated.

`make_sequence()`
: Converts the response iterator in a list. By default this happens automatically if required. If *implicit_sequence_conversion* is disabled, this method is not automatically called and some properties might raise exceptions. This also encodes all the items.

  New in version 0.6.

`set_cookie(`*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=False*`)`
: Sets a cookie. The parameters are the same as in the cookie *Morsel* object in the Python standard library but it accepts unicode data, too.

  Parameters

  - `key` – the key (name) of the cookie to be set.
  - `value` – the value of the cookie.
  - `max_age` – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client's browser session.
  - `expires` – should be a *datetime* object or UNIX timestamp.
  - `domain` – if you want to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domain `www.example.com`, `foo.example.com` etc. Otherwise, a cookie will only be readable by the domain that set it.
  - `path` – limits the cookie to a given path, per default it will span the whole domain.

`set_data(`*value*`)`
: Sets a new string as response. The value set must either by a unicode or bytestring. If a unicode string is set it's encoded automatically to the charset of the response (utf-8 by default).

  New in version 0.9.

`status`
: The HTTP Status code

`status_code`
: The HTTP Status code as number

10.4 Mixin Classes

Werkzeug also provides helper mixins for various HTTP related functionality such as etags, cache control, user agents etc. When subclassing you can mix those classes in to extend the functionality of the *BaseRequest* or *BaseResponse* object. Here a small example for a request object that parses accept headers:

```
from werkzeug.wrappers import AcceptMixin, BaseRequest

class Request(BaseRequest, AcceptMixin):
    pass
```

The *Request* and *Response* classes subclass the *BaseRequest* and *BaseResponse* classes and implement all the mixins Werkzeug provides:

class `werkzeug.wrappers.Request`(*environ*, *populate_request=True*, *shallow=False*)
: Full featured request object implementing the following mixins:

  - *AcceptMixin* for accept header parsing
  - *ETagRequestMixin* for etag and cache control handling
  - *UserAgentMixin* for user agent introspection
  - *AuthorizationMixin* for http auth handling
  - *CommonRequestDescriptorsMixin* for common headers

class `werkzeug.wrappers.Response`(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)
: Full featured response object implementing the following mixins:

  - *ETagResponseMixin* for etag and cache control handling
  - *ResponseStreamMixin* to add support for the *stream* property
  - *CommonResponseDescriptorsMixin* for various HTTP descriptors
  - *WWWAuthenticateMixin* for HTTP authentication support

class `werkzeug.wrappers.AcceptMixin`
: A mixin for classes with an `environ` attribute to get all the HTTP accept headers as *Accept* objects (or subclasses thereof).

  `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.

class `werkzeug.wrappers.AuthorizationMixin`
: Adds an *`authorization`* property that represents the parsed value of the *Authorization* header as *`Authorization`* object.

`authorization`
: The *Authorization* object in parsed form.

class `werkzeug.wrappers.ETagRequestMixin`
: Add entity tag and cache descriptors to a request object or object with a WSGI environment available as *`environ`*. This not only provides access to etags but also to the cache control header.

`cache_control`
: A *`RequestCacheControl`* object for the incoming cache control headers.

`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.

`range`
: The parsed *Range* header.

  New in version 0.7.

  Return type *`Range`*

class `werkzeug.wrappers.ETagResponseMixin`
: Adds extra functionality to a response object for etag and cache handling. This

mixin requires an object with at least a *headers* object that implements a dict like interface similar to *Headers*.

If you want the *freeze()* method to automatically add an etag, you have to mixin this method before the response base class. The default response class does not do that.

`accept_ranges`
: The *Accept-Ranges* header. Even though the name would indicate that multiple values are supported, it must be one string token only.

  The values `'bytes'` and `'none'` are common.

  New in version 0.7.

`add_etag`(*overwrite=False*, *weak=False*)
: Add an etag for the current response if there is none yet.

`cache_control`
: The Cache-Control general-header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain.

`content_range`
: The *Content-Range* header as *ContentRange* object. Even if the header is not set it wil provide such an object for easier manipulation.

  New in version 0.7.

`freeze`(*no_etag=False*)
: Call this method if you want to make your response object ready for pickeling. This buffers the generator if there is one. This also sets the etag unless *no_etag* is set to *True*.

`get_etag()`
: Return a tuple in the form `(etag, is_weak)`. If there is no ETag the return value is `(None, None)`.

`make_conditional`(*request_or_environ*)
: Make the response conditional to the request. This method works best if an etag was defined for the response already. The *add_etag* method can be used to do that. If called without etag just the date header is set.

  This does nothing if the request method in the request or environ is anything but GET or HEAD.

  It does not remove the body of the response because that's something the `__call__()` function does for us automatically.

  Returns self so that you can do `return resp.make_conditional(req)` but modifies the object in-place.

  Parameters `request_or_environ` – a request object or WSGI environment to be used to make the response conditional against.

set_etag(*etag*, *weak=False*)
Set the etag, and override the old one if there was one.

class werkzeug.wrappers.ResponseStreamMixin
Mixin for *BaseRequest* subclasses. Classes that inherit from this mixin will automatically get a *stream* property that provides a write-only interface to the response iterable.

stream
The response iterable as write-only stream.

class werkzeug.wrappers.CommonRequestDescriptorsMixin
A mixin for *BaseRequest* subclasses. Request objects that mix this class in will automatically get descriptors for a couple of HTTP headers with automatic type conversion.

New in version 0.5.

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.

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.

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.

`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'}`.

`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.

`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).

class `werkzeug.wrappers.CommonResponseDescriptorsMixin`

A mixin for *BaseResponse* subclasses. Response objects that mix this class in will automatically get descriptors for a couple of HTTP headers with automatic type conversion.

`age`
: The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server.

  Age values are non-negative decimal integers, representing time in seconds.

`allow`
: The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. An Allow header field MUST be present in a 405 (Method Not Allowed) response.

`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.

`content_language`
: The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this might not be equivalent to all the languages used within the entity-body.

`content_length`
: The Content-Length entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient 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_location`
: The Content-Location entity-header field MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI.

`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.)

`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.

`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.

`expires`
: The Expires entity-header field gives the date/time after which the response is considered stale. A stale cache entry may not normally be returned by a cache.

`last_modified`
: The Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified.

`location`
: The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.

`mimetype`
: The mimetype (content type without charset etc.)

`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'}`.

  New in version 0.5.

`retry_after`
: The Retry-After response-header field can be used with a 503 (Service Un-

available) response to indicate how long the service is expected to be unavailable to the requesting client.

Time in seconds until expiration or date.

`vary`
: The Vary field value indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation.

class `werkzeug.wrappers.WWWAuthenticateMixin`
: Adds a *`www_authenticate`* property to a response object.

`www_authenticate`
: The *WWW-Authenticate* header in a parsed form.

class `werkzeug.wrappers.UserAgentMixin`
: Adds a *user_agent* attribute to the request object which contains the parsed user agent of the browser that triggered the request as a *`UserAgent`* object.

`user_agent`
: The current user agent.

CHAPTER

ELEVEN

URL ROUTING

When it comes to combining multiple controller or view functions (however you want to call them), you need a dispatcher. A simple way would be applying regular expression tests on `PATH_INFO` and call registered callback functions that return the value.

Werkzeug provides a much more powerful system, similar to Routes. All the objects mentioned on this page must be imported from *werkzeug.routing*, not from *werkzeug*!

11.1 Quickstart

Here is a simple example which could be the URL definition for a blog:

```
from werkzeug.routing import Map, Rule, NotFound, RequestRedirect

url_map = Map([
    Rule('/', endpoint='blog/index'),
    Rule('/<int:year>/', endpoint='blog/archive'),
    Rule('/<int:year>/<int:month>/', endpoint='blog/archive'),
    Rule('/<int:year>/<int:month>/<int:day>/', endpoint='blog/archive'),
    Rule('/<int:year>/<int:month>/<int:day>/<slug>',
         endpoint='blog/show_post'),
    Rule('/about', endpoint='blog/about_me'),
    Rule('/feeds/', endpoint='blog/feeds'),
    Rule('/feeds/<feed_name>.rss', endpoint='blog/show_feed')
])

def application(environ, start_response):
    urls = url_map.bind_to_environ(environ)
    try:
        endpoint, args = urls.match()
    except HTTPException, e:
        return e(environ, start_response)
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Rule points to %r with arguments %r' % (endpoint, args)]
```

So what does that do? First of all we create a new *Map* which stores a bunch of URL rules. Then we pass it a list of *Rule* objects.

Each *Rule* object is instantiated with a string that represents a rule and an endpoint which will be the alias for what view the rule represents. Multiple rules can have the same endpoint, but should have different arguments to allow URL construction.

The format for the URL rules is straightforward, but explained in detail below.

Inside the WSGI application we bind the url_map to the current request which will return a new *MapAdapter*. This url_map adapter can then be used to match or build domains for the current request.

The *MapAdapter.match()* method can then either return a tuple in the form `(endpoint, args)` or raise one of the three exceptions *NotFound*, *MethodNotAllowed*, or `RequestRedirect`. For more details about those exceptions have a look at the documentation of the *MapAdapter.match()* method.

11.2 Rule Format

Rule strings basically are just normal URL paths with placeholders in the format `<converter(arguments):name>`, where converter and the arguments are optional. If no converter is defined, the *default* converter is used (which means *string* in the normal configuration).

URL rules that end with a slash are branch URLs, others are leaves. If you have *strict_slashes* enabled (which is the default), all branch URLs that are visited without a trailing slash will trigger a redirect to the same URL with that slash appended.

The list of converters can be extended, the default converters are explained below.

11.3 Builtin Converters

Here a list of converters that come with Werkzeug:

class `werkzeug.routing.UnicodeConverter`(*map*, *minlength=1*, *maxlength=None*, *length=None*)

This converter is the default converter and accepts any string but only one path segment. Thus the string can not include a slash.

This is the default validator.

Example:

```
Rule('/pages/<page>'),
Rule('/<string(length=2):lang_code>')
```

Parameters

- `map` – the *Map*.

- `minlength` – the minimum length of the string. Must be greater or equal 1.
- `maxlength` – the maximum length of the string.
- `length` – the exact length of the string.

class `werkzeug.routing.PathConverter`(*map*)
Like the default *`UnicodeConverter`*, but it also matches slashes. This is useful for wikis and similar applications:

```
Rule('/<path:wikipage>')
Rule('/<path:wikipage>/edit')
```

Parameters `map` – the *`Map`*.

class `werkzeug.routing.AnyConverter`(*map*, **items*)
Matches one of the items provided. Items can either be Python identifiers or strings:

```
Rule('/<any(about, help, imprint, class, "foo,bar"):page_name>')
```

Parameters

- `map` – the *`Map`*.
- `items` – this function accepts the possible items as positional arguments.

class `werkzeug.routing.IntegerConverter`(*map*, *fixed_digits=0*, *min=None*, *max=None*)
This converter only accepts integer values:

```
Rule('/page/<int:page>')
```

This converter does not support negative values.

Parameters

- `map` – the *`Map`*.
- `fixed_digits` – the number of fixed digits in the URL. If you set this to `4` for example, the application will only match if the url looks like `/0001/`. The default is variable length.
- `min` – the minimal value.
- `max` – the maximal value.

class `werkzeug.routing.FloatConverter`(*map*, *min=None*, *max=None*)
This converter only accepts floating point values:

```
Rule('/probability/<float:probability>')
```

This converter does not support negative values.

Parameters

- `map` – the *Map*.
- `min` – the minimal value.
- `max` – the maximal value.

class `werkzeug.routing.UUIDConverter`(*map*)
This converter only accepts UUID strings:

```
Rule('/object/<uuid:identifier>')
```

New in version 0.10.

Parameters `map` – the *Map*.

11.4 Maps, Rules and Adapters

class `werkzeug.routing.Map`(*rules=None*, *default_subdomain=''*, *charset='utf-8'*, *strict_slashes=True*, *redirect_defaults=True*, *converters=None*, *sort_parameters=False*, *sort_key=None*, *encoding_errors='replace'*, *host_matching=False*)
The map class stores all the URL rules and some configuration parameters. Some of the configuration values are only stored on the *Map* instance since those affect all rules, others are just defaults and can be overridden for each rule. Note that you have to specify all arguments besides the *rules* as keyword arguments!

Parameters

- `rules` – sequence of url rules for this map.
- `default_subdomain` – The default subdomain for rules without a subdomain defined.
- `charset` – charset of the url. defaults to `"utf-8"`
- `strict_slashes` – Take care of trailing slashes.
- `redirect_defaults` – This will redirect to the default rule if it wasn't visited that way. This helps creating unique URLs.
- `converters` – A dict of converters that adds additional converters to the list of converters. If you redefine one converter this will override the original one.
- `sort_parameters` – If set to *True* the url parameters are sorted. See *url_encode* for more details.
- `sort_key` – The sort key function for *url_encode*.
- `encoding_errors` – the error method to use for decoding

- `host_matching` – if set to *True* it enables the host matching feature and disables the subdomain one. If enabled the *host* parameter to rules is used instead of the *subdomain* one.

New in version 0.5: *sort_parameters* and *sort_key* was added.

New in version 0.7: *encoding_errors* and *host_matching* was added.

`converters`
: The dictionary of converters. This can be modified after the class was created, but will only affect rules added after the modification. If the rules are defined with the list passed to the class, the *converters* parameter to the constructor has to be used instead.

`add`(*rulefactory*)
: Add a new rule or factory to the map and bind it. Requires that the rule is not bound to another map.

  Parameters `rulefactory` – a *`Rule`* or *`RuleFactory`*

`bind`(*server_name*, *script_name=None*, *subdomain=None*, *url_scheme='http'*, *default_method='GET'*, *path_info=None*, *query_args=None*)
: Return a new *`MapAdapter`* with the details specified to the call. Note that *script_name* will default to `'/'` if not further specified or *None*. The *server_name* at least is a requirement because the HTTP RFC requires absolute URLs for redirects and so all redirect exceptions raised by Werkzeug will contain the full canonical URL.

  If no path_info is passed to `match()` it will use the default path info passed to bind. While this doesn't really make sense for manual bind calls, it's useful if you bind a map to a WSGI environment which already contains the path info.

  subdomain will default to the *default_subdomain* for this map if no defined. If there is no *default_subdomain* you cannot use the subdomain feature.

  New in version 0.7: *query_args* added

  New in version 0.8: *query_args* can now also be a string.

`bind_to_environ`(*environ*, *server_name=None*, *subdomain=None*)
: Like *`bind()`* but you can pass it an WSGI environment and it will fetch the information from that dictionary. Note that because of limitations in the protocol there is no way to get the current subdomain and real *server_name* from the environment. If you don't provide it, Werkzeug will use *SERVER_NAME* and *SERVER_PORT* (or *HTTP_HOST* if provided) as used *server_name* with disabled subdomain feature.

  If *subdomain* is *None* but an environment and a server name is provided it will calculate the current subdomain automatically. Example: *server_name* is `'example.com'` and the *SERVER_NAME* in the wsgi *environ* is `'staging.dev.example.com'` the calculated subdomain will be `'staging.dev'`.

If the object passed as environ has an environ attribute, the value of this attribute is used instead. This allows you to pass request objects. Additionally *PATH_INFO* added as a default of the *MapAdapter* so that you don't have to pass the path info to the match method.

Changed in version 0.5: previously this method accepted a bogus *calculate_subdomain* parameter that did not have any effect. It was removed because of that.

Changed in version 0.8: This will no longer raise a ValueError when an unexpected server name was passed.

Parameters

- `environ` – a WSGI environment.
- `server_name` – an optional server name hint (see above).
- `subdomain` – optionally the current subdomain (see above).

`default_converters` **= ImmutableDict({'uuid': <class 'werkzeug.routing.UUIDConverter'>,**

New in version 0.6: a dict of default converters to be used.

`is_endpoint_expecting`(*endpoint*, **arguments*)

Iterate over all rules and check if the endpoint expects the arguments provided. This is for example useful if you have some URLs that expect a language code and others that do not and you want to wrap the builder a bit so that the current language code is automatically added if not provided but endpoints expect it.

Parameters

- `endpoint` – the endpoint to check.
- `arguments` – this function accepts one or more arguments as positional arguments. Each one of them is checked.

`iter_rules`(*endpoint=None*)

Iterate over all rules or the rules of an endpoint.

Parameters `endpoint` – if provided only the rules for that endpoint are returned.

Returns an iterator

`update`()

Called before matching and building to keep the compiled rules in the correct order after things changed.

class `werkzeug.routing.MapAdapter`(*map*, *server_name*, *script_name*, *subdomain*, *url_scheme*, *path_info*, *default_method*, *query_args=None*)

Returned by *Map.bind()* or *Map.bind_to_environ()* and does the URL matching and building based on runtime information.

`allowed_methods`(*path_info=None*)

Returns the valid methods that match for a given path.

New in version 0.7.

`build`(*endpoint*, *values=None*, *method=None*, *force_external=False*, *append_unknown=True*)

Building URLs works pretty much the other way round. Instead of *match* you call *build* and pass it the endpoint and a dict of arguments for the placeholders.

The *build* function also accepts an argument called *force_external* which, if you set it to *True* will force external URLs. Per default external URLs (include the server name) will only be used if the target URL is on a different subdomain.

```
>>> m = Map([
...     Rule('/', endpoint='index'),
...     Rule('/downloads/', endpoint='downloads/index'),
...     Rule('/downloads/<int:id>', endpoint='downloads/show')
... ])
>>> urls = m.bind("example.com", "/")
>>> urls.build("index", {})
'/'
>>> urls.build("downloads/show", {'id': 42})
'/downloads/42'
>>> urls.build("downloads/show", {'id': 42}, force_external=True)
'http://example.com/downloads/42'
```

Because URLs cannot contain non ASCII data you will always get bytestrings back. Non ASCII characters are urlencoded with the charset defined on the map instance.

Additional values are converted to unicode and appended to the URL as URL querystring parameters:

```
>>> urls.build("index", {'q': 'My Searchstring'})
'/?q=My+Searchstring'
```

When processing those additional values, lists are furthermore interpreted as multiple values (as per *werkzeug.datastructures.MultiDict*):

```
>>> urls.build("index", {'q': ['a', 'b', 'c']})
'/?q=a&q=b&q=c'
```

If a rule does not exist when building a *BuildError* exception is raised.

The build method accepts an argument called *method* which allows you to specify the method you want to have an URL built for if you have different methods for the same endpoint specified.

New in version 0.6: the *append_unknown* parameter was added.

Parameters

- `endpoint` – the endpoint of the URL to build.

- `values` – the values for the URL to build. Unhandled values are appended to the URL as query parameters.
- `method` – the HTTP method for the rule if there are different URLs for different methods on the same endpoint.
- `force_external` – enforce full canonical external URLs. If the URL scheme is not provided, this will generate a protocol-relative URL.
- `append_unknown` – unknown parameters are appended to the generated URL as query string argument. Disable this if you want the builder to ignore those.

`dispatch`(*view_func*, *path_info=None*, *method=None*, *catch_http_exceptions=False*)

Does the complete dispatching process. *view_func* is called with the endpoint and a dict with the values for the view. It should look up the view function, call it, and return a response object or WSGI application. http exceptions are not caught by default so that applications can display nicer error messages by just catching them by hand. If you want to stick with the default error messages you can pass it `catch_http_exceptions=True` and it will catch the http exceptions.

Here a small example for the dispatch usage:

```
from werkzeug.wrappers import Request, Response
from werkzeug.wsgi import responder
from werkzeug.routing import Map, Rule

def on_index(request):
    return Response('Hello from the index')

url_map = Map([Rule('/', endpoint='index')])
views = {'index': on_index}

@responder
def application(environ, start_response):
    request = Request(environ)
    urls = url_map.bind_to_environ(environ)
    return urls.dispatch(lambda e, v: views[e](request, **v),
                         catch_http_exceptions=True)
```

Keep in mind that this method might return exception objects, too, so use `Response.force_type` to get a response object.

Parameters

- `view_func` – a function that is called with the endpoint as first argument and the value dict as second. Has to dispatch to the actual view function with this information. (see above)
- `path_info` – the path info to use for matching. Overrides the path info specified on binding.

- `method` – the HTTP method used for matching. Overrides the method specified on binding.
- `catch_http_exceptions` – set to *True* to catch any of the werkzeug `HTTPExceptions`.

`get_default_redirect`(*rule*, *method*, *values*, *query_args*)
A helper that returns the URL to redirect to if it finds one. This is used for default redirecting only.

Internal

`get_host`(*domain_part*)
Figures out the full host name for the given domain part. The domain part is a subdomain in case host matching is disabled or a full host name.

`make_alias_redirect_url`(*path*, *endpoint*, *values*, *method*, *query_args*)
Internally called to make an alias redirect URL.

`make_redirect_url`(*path_info*, *query_args=None*, *domain_part=None*)
Creates a redirect URL.

Internal

`match`(*path_info=None*, *method=None*, *return_rule=False*, *query_args=None*)
The usage is simple: you just pass the match method the current path info as well as the method (which defaults to *GET*). The following things can then happen:

- you receive a *NotFound* exception that indicates that no URL is matching. A *NotFound* exception is also a WSGI application you can call to get a default page not found page (happens to be the same object as *werkzeug.exceptions.NotFound*)
- you receive a *MethodNotAllowed* exception that indicates that there is a match for this URL but not for the current request method. This is useful for RESTful applications.
- you receive a *RequestRedirect* exception with a *new_url* attribute. This exception is used to notify you about a request Werkzeug requests from your WSGI application. This is for example the case if you request `/foo` although the correct URL is `/foo/` You can use the *RequestRedirect* instance as response-like object similar to all other subclasses of *HTTPException*.
- you get a tuple in the form `(endpoint, arguments)` if there is a match (unless *return_rule* is True, in which case you get a tuple in the form `(rule, arguments)`)

If the path info is not passed to the match method the default path info of the map is used (defaults to the root URL if not defined explicitly).

All of the exceptions raised are subclasses of *HTTPException* so they can be used as WSGI responses. The will all render generic error or redirect pages.

Here is a small example for matching:

```
>>> m = Map([
...     Rule('/', endpoint='index'),
...     Rule('/downloads/', endpoint='downloads/index'),
...     Rule('/downloads/<int:id>', endpoint='downloads/show')
... ])
>>> urls = m.bind("example.com", "/")
>>> urls.match("/", "GET")
('index', {})
>>> urls.match("/downloads/42")
('downloads/show', {'id': 42})
```

And here is what happens on redirect and missing URLs:

```
>>> urls.match("/downloads")
Traceback (most recent call last):
  ...
RequestRedirect: http://example.com/downloads/
>>> urls.match("/missing")
Traceback (most recent call last):
  ...
NotFound: 404 Not Found
```

Parameters

- `path_info` – the path info to use for matching. Overrides the path info specified on binding.
- `method` – the HTTP method used for matching. Overrides the method specified on binding.
- `return_rule` – return the rule that matched instead of just the endpoint (defaults to *False*).
- `query_args` – optional query arguments that are used for automatic redirects as string or dictionary. It's currently not possible to use the query arguments for URL matching.

New in version 0.6: *return_rule* was added.

New in version 0.7: *query_args* was added.

Changed in version 0.8: *query_args* can now also be a string.

`test`(*path_info=None*, *method=None*)

Test if a rule would match. Works like *match* but returns *True* if the URL matches, or *False* if it does not exist.

Parameters

- `path_info` – the path info to use for matching. Overrides the path info specified on binding.

- `method` – the HTTP method used for matching. Overrides the method specified on binding.

class `werkzeug.routing.Rule`(*string*, *defaults=None*, *subdomain=None*, *methods=None*, *build_only=False*, *endpoint=None*, *strict_slashes=None*, *redirect_to=None*, *alias=False*, *host=None*)

A Rule represents one URL pattern. There are some options for *Rule* that change the way it behaves and are passed to the *Rule* constructor. Note that besides the rule-string all arguments *must* be keyword arguments in order to not break the application on Werkzeug upgrades.

string Rule strings basically are just normal URL paths with placeholders in the format `<converter(arguments):name>` where the converter and the arguments are optional. If no converter is defined the *default* converter is used which means *string* in the normal configuration.

URL rules that end with a slash are branch URLs, others are leaves. If you have *strict_slashes* enabled (which is the default), all branch URLs that are matched without a trailing slash will trigger a redirect to the same URL with the missing slash appended.

The converters are defined on the *Map*.

endpoint The endpoint for this rule. This can be anything. A reference to a function, a string, a number etc. The preferred way is using a string because the endpoint is used for URL generation.

defaults An optional dict with defaults for other rules with the same endpoint. This is a bit tricky but useful if you want to have unique URLs:

```
url_map = Map([
    Rule('/all/', defaults={'page': 1}, endpoint='all_entries'),
    Rule('/all/page/<int:page>', endpoint='all_entries')
])
```

If a user now visits `http://example.com/all/page/1` he will be redirected to `http://example.com/all/`. If *redirect_defaults* is disabled on the *Map* instance this will only affect the URL generation.

subdomain The subdomain rule string for this rule. If not specified the rule only matches for the *default_subdomain* of the map. If the map is not bound to a subdomain this feature is disabled.

Can be useful if you want to have user profiles on different subdomains and all subdomains are forwarded to your application:

```
url_map = Map([
    Rule('/', subdomain='<username>', endpoint='user/homepage'),
    Rule('/stats', subdomain='<username>', endpoint='user/stats')
])
```

methods A sequence of http methods this rule applies to. If not specified, all methods are allowed. For example this can be useful if you want different

endpoints for *POST* and *GET*. If methods are defined and the path matches but the method matched against is not in this list or in the list of another rule for that path the error raised is of the type *MethodNotAllowed* rather than *NotFound*. If *GET* is present in the list of methods and *HEAD* is not, *HEAD* is added automatically.

Changed in version 0.6.1: *HEAD* is now automatically added to the methods if *GET* is present. The reason for this is that existing code often did not work properly in servers not rewriting *HEAD* to *GET* automatically and it was not documented how *HEAD* should be treated. This was considered a bug in Werkzeug because of that.

strict_slashes Override the *Map* setting for *strict_slashes* only for this rule. If not specified the *Map* setting is used.

build_only Set this to True and the rule will never match but will create a URL that can be build. This is useful if you have resources on a subdomain or folder that are not handled by the WSGI application (like static data)

redirect_to If given this must be either a string or callable. In case of a callable it's called with the url adapter that triggered the match and the values of the URL as keyword arguments and has to return the target for the redirect, otherwise it has to be a string with placeholders in rule syntax:

```
def foo_with_slug(adapter, id):
    # ask the database for the slug for the old id.  this of
    # course has nothing to do with werkzeug.
    return 'foo/' + Foo.get_slug_for_id(id)

url_map = Map([
    Rule('/foo/<slug>', endpoint='foo'),
    Rule('/some/old/url/<slug>', redirect_to='foo/<slug>'),
    Rule('/other/old/url/<int:id>', redirect_to=foo_with_slug)
])
```

When the rule is matched the routing system will raise a *RequestRedirect* exception with the target for the redirect.

Keep in mind that the URL will be joined against the URL root of the script so don't use a leading slash on the target URL unless you really mean root of that domain.

alias If enabled this rule serves as an alias for another rule with the same endpoint and arguments.

host If provided and the URL map has host matching enabled this can be used to provide a match rule for the whole host. This also means that the subdomain feature is disabled.

New in version 0.7: The *alias* and *host* parameters were added.

`empty()`
Return an unbound copy of this rule.

This can be useful if want to reuse an already bound URL for another map. See `get_empty_kwargs` to override what keyword arguments are provided to the new copy.

11.5 Rule Factories

class `werkzeug.routing.RuleFactory`

As soon as you have more complex URL setups it's a good idea to use rule factories to avoid repetitive tasks. Some of them are builtin, others can be added by subclassing *RuleFactory* and overriding *get_rules*.

`get_rules(`*map*`)`

Subclasses of *RuleFactory* have to override this method and return an iterable of rules.

class `werkzeug.routing.Subdomain(`*subdomain*, *rules*`)`

All URLs provided by this factory have the subdomain set to a specific domain. For example if you want to use the subdomain for the current language this can be a good setup:

```
url_map = Map([
    Rule('/', endpoint='#select_language'),
    Subdomain('<string(length=2):lang_code>', [
        Rule('/', endpoint='index'),
        Rule('/about', endpoint='about'),
        Rule('/help', endpoint='help')
    ])
])
```

All the rules except for the `'#select_language'` endpoint will now listen on a two letter long subdomain that holds the language code for the current request.

class `werkzeug.routing.Submount(`*path*, *rules*`)`

Like *Subdomain* but prefixes the URL rule with a given string:

```
url_map = Map([
    Rule('/', endpoint='index'),
    Submount('/blog', [
        Rule('/', endpoint='blog/index'),
        Rule('/entry/<entry_slug>', endpoint='blog/show')
    ])
])
```

Now the rule `'blog/show'` matches `/blog/entry/<entry_slug>`.

class `werkzeug.routing.EndpointPrefix(`*prefix*, *rules*`)`

Prefixes all endpoints (which must be strings for this factory) with another string. This can be useful for sub applications:

```
url_map = Map([
    Rule('/', endpoint='index'),
```

```
        EndpointPrefix('blog/', [Submount('/blog', [
            Rule('/', endpoint='index'),
            Rule('/entry/<entry_slug>', endpoint='show')
        ])])
    ])
```

11.6 Rule Templates

class werkzeug.routing.RuleTemplate(*rules*)

Returns copies of the rules wrapped and expands string templates in the endpoint, rule, defaults or subdomain sections.

Here a small example for such a rule template:

```
from werkzeug.routing import Map, Rule, RuleTemplate

resource = RuleTemplate([
    Rule('/$name/', endpoint='$name.list'),
    Rule('/$name/<int:id>', endpoint='$name.show')
])

url_map = Map([resource(name='user'), resource(name='page')])
```

When a rule template is called the keyword arguments are used to replace the placeholders in all the string parameters.

11.7 Custom Converters

You can easily add custom converters. The only thing you have to do is to subclass BaseConverter and pass that new converter to the url_map. A converter has to provide two public methods: *to_python* and *to_url*, as well as a member that represents a regular expression. Here is a small example:

```
from random import randrange
from werkzeug.routing import Rule, Map, BaseConverter, ValidationError

class BooleanConverter(BaseConverter):

    def __init__(self, url_map, randomify=False):
        super(BooleanConverter, self).__init__(url_map)
        self.randomify = randomify
        self.regex = '(?:yes|no|maybe)'

    def to_python(self, value):
        if value == 'maybe':
            if self.randomify:
                return not randrange(2)
```

```
            raise ValidationError()
        return value == 'yes'

    def to_url(self, value):
        return value and 'yes' or 'no'

url_map = Map([
    Rule('/vote/<bool:werkzeug_rocks>', endpoint='vote'),
    Rule('/vote/<bool(randomify=True):foo>', endpoint='foo')
], converters={'bool': BooleanConverter})
```

If you want that converter to be the default converter, name it `'default'`.

11.8 Host Matching

New in version 0.7.

Starting with Werkzeug 0.7 it's also possible to do matching on the whole host names instead of just the subdomain. To enable this feature you need to pass `host_matching=True` to the *Map* constructor and provide the *host* argument to all routes:

```
url_map = Map([
    Rule('/', endpoint='www_index', host='www.example.com'),
    Rule('/', endpoint='help_index', host='help.example.com')
], host_matching=True)
```

Variable parts are of course also possible in the host section:

```
url_map = Map([
    Rule('/', endpoint='www_index', host='www.example.com'),
    Rule('/', endpoint='user_index', host='<user>.example.com')
], host_matching=True)
```

CHAPTER

TWELVE

WSGI HELPERS

The following classes and functions are designed to make working with the WSGI specification easier or operate on the WSGI layer. All the functionality from this module is available on the high-level *Request/Response classes*.

12.1 Iterator / Stream Helpers

These classes and functions simplify working with the WSGI application iterator and the input stream.

class `werkzeug.wsgi.ClosingIterator`(*iterable*, *callbacks=None*)

The WSGI specification requires that all middlewares and gateways respect the *close* callback of an iterator. Because it is useful to add another close action to a returned iterator and adding a custom iterator is a boring task this class can be used for that:

```
return ClosingIterator(app(environ, start_response), [cleanup_session,
                                                      cleanup_locals])
```

If there is just one close function it can be passed instead of the list.

A closing iterator is not needed if the application uses response objects and finishes the processing if the response is started:

```
try:
    return response(environ, start_response)
finally:
    cleanup_session()
    cleanup_locals()
```

class `werkzeug.wsgi.FileWrapper`(*file*, *buffer_size=8192*)

This class can be used to convert a `file`-like object into an iterable. It yields *buffer_size* blocks until the file is fully read.

You should not use this class directly but rather use the *`wrap_file()`* function that uses the WSGI server's file wrapper support if it's available.

New in version 0.5.

If you're using this object together with a `BaseResponse` you have to use the *direct_passthrough* mode.

Parameters

- `file` – a `file`-like object with a `read()` method.
- `buffer_size` – number of bytes for one iteration.

class `werkzeug.wsgi.LimitedStream(`*stream, limit*`)`

Wraps a stream so that it doesn't read more than n bytes. If the stream is exhausted and the caller tries to get more bytes from it *on_exhausted()* is called which by default returns an empty string. The return value of that function is forwarded to the reader function. So if it returns an empty string *read()* will return an empty string as well.

The limit however must never be higher than what the stream can output. Otherwise *readlines()* will try to read past the limit.

Note on WSGI compliance

calls to *readline()* and *readlines()* are not WSGI compliant because it passes a size argument to the readline methods. Unfortunately the WSGI PEP is not safely implementable without a size argument to *readline()* because there is no EOF marker in the stream. As a result of that the use of *readline()* is discouraged.

For the same reason iterating over the *LimitedStream* is not portable. It internally calls *readline()*.

We strongly suggest using *read()* only or using the *make_line_iter()* which safely iterates line-based over a WSGI input stream.

Parameters

- `stream` – the stream to wrap.
- `limit` – the limit for the stream, must not be longer than what the string can provide if the stream does not end with *EOF* (like *wsgi.input*)

`exhaust(`*chunk_size=65536*`)`

Exhaust the stream. This consumes all the data left until the limit is reached.

Parameters `chunk_size` – the size for a chunk. It will read the chunk until the stream is exhausted and throw away the results.

`is_exhausted`

If the stream is exhausted this attribute is *True*.

`on_disconnect()`

What should happen if a disconnect is detected? The return value of this function is returned from read functions in case the client went away. By default a *ClientDisconnected* exception is raised.

`on_exhausted()`
: This is called when the stream tries to read past the limit. The return value of this function is returned from the reading function.

`read(`*size=None*`)`
: Read *size* bytes or if size is not provided everything is read.

  Parameters `size` – the number of bytes read.

`readline(`*size=None*`)`
: Reads one line from the stream.

`readlines(`*size=None*`)`
: Reads a file into a list of strings. It calls *`readline()`* until the file is read to the end. It does support the optional *size* argument if the underlaying stream supports it for *readline*.

`tell()`
: Returns the position of the stream.

  New in version 0.9.

`werkzeug.wsgi.make_line_iter(`*stream, limit=None, buffer_size=10240*`)`
: Safely iterates line-based over an input stream. If the input stream is not a *`LimitedStream`* the *limit* parameter is mandatory.

  This uses the stream's `read()` method internally as opposite to the `readline()` method that is unsafe and can only be used in violation of the WSGI specification. The same problem applies to the *__iter__* function of the input stream which calls `readline()` without arguments.

  If you need line-by-line processing it's strongly recommended to iterate over the input stream using this helper function.

  Changed in version 0.8: This function now ensures that the limit was reached.

  New in version 0.9: added support for iterators as input stream.

  Parameters

  - `stream` – the stream or iterate to iterate over.
  - `limit` – the limit in bytes for the stream. (Usually content length. Not necessary if the *stream* is a *`LimitedStream`*.
  - `buffer_size` – The optional buffer size.

`werkzeug.wsgi.make_chunk_iter(`*stream, separator, limit=None, buffer_size=10240*`)`
: Works like *`make_line_iter()`* but accepts a separator which divides chunks. If you want newline based processing you should use *`make_line_iter()`* instead as it supports arbitrary newline markers.

  New in version 0.8.

  New in version 0.9: added support for iterators as input stream.

  Parameters

- `stream` – the stream or iterate to iterate over.
- `separator` – the separator that divides chunks.
- `limit` – the limit in bytes for the stream. (Usually content length. Not necessary if the *stream* is otherwise already limited).
- `buffer_size` – The optional buffer size.

`werkzeug.wsgi.wrap_file(`*environ*, *file*, *buffer_size=8192*`)`

Wraps a file. This uses the WSGI server's file wrapper if available or otherwise the generic *`FileWrapper`*.

New in version 0.5.

If the file wrapper from the WSGI server is used it's important to not iterate over it from inside the application but to pass it through unchanged. If you want to pass out a file wrapper inside a response object you have to set `direct_passthrough` to *True*.

More information about file wrappers are available in **PEP 333**.

Parameters

- `file` – a `file`-like object with a `read()` method.
- `buffer_size` – number of bytes for one iteration.

12.2 Environ Helpers

These functions operate on the WSGI environment. They extract useful information or perform common manipulations:

`werkzeug.wsgi.get_host(`*environ*, *trusted_hosts=None*`)`

Return the real host for the given WSGI environment. This first checks the *X-Forwarded-Host* header, then the normal *Host* header, and finally the *SERVER_NAME* environment variable (using the first one it finds).

Optionally it verifies that the host is in a list of trusted hosts. If the host is not in there it will raise a *`SecurityError`*.

Parameters

- `environ` – the WSGI environment to get the host of.
- `trusted_hosts` – a list of trusted hosts, see *`host_is_trusted()`* for more information.

`werkzeug.wsgi.get_content_length(`*environ*`)`

Returns the content length from the WSGI environment as integer. If it's not available *None* is returned.

New in version 0.9.

Parameters `environ` – the WSGI environ to fetch the content length from.

`werkzeug.wsgi.get_input_stream(`*environ*, *safe_fallback=True*`)`

Returns the input stream from the WSGI environment and wraps it in the most sensible way possible. The stream returned is not the raw WSGI stream in most cases but one that is safe to read from without taking into account the content length.

New in version 0.9.

Parameters

- `environ` – the WSGI environ to fetch the stream from.
- `safe` – indicates weather the function should use an empty stream as safe fallback or just return the original WSGI input stream if it can't wrap it safely. The default is to return an empty string in those cases.

`werkzeug.wsgi.get_current_url(`*environ*, *root_only=False*, *strip_querystring=False*, *host_only=False*, *trusted_hosts=None*`)`

A handy helper function that recreates the full URL as IRI for the current request or parts of it. Here an example:

```
>>> from werkzeug.test import create_environ
>>> env = create_environ("/?param=foo", "http://localhost/script")
>>> get_current_url(env)
'http://localhost/script/?param=foo'
>>> get_current_url(env, root_only=True)
'http://localhost/'
>>> get_current_url(env, host_only=True)
'http://localhost/'
>>> get_current_url(env, strip_querystring=True)
'http://localhost/script/'
```

This optionally it verifies that the host is in a list of trusted hosts. If the host is not in there it will raise a *SecurityError*.

Note that the string returned might contain unicode characters as the representation is an IRI not an URI. If you need an ASCII only representation you can use the *iri_to_uri()* function:

```
>>> from werkzeug.urls import iri_to_uri
>>> iri_to_uri(get_current_url(env))
'http://localhost/script/?param=foo'
```

Parameters

- `environ` – the WSGI environment to get the current URL from.
- `root_only` – set *True* if you only want the root URL.

- `strip_querystring` – set to *True* if you don't want the querystring.
- `host_only` – set to *True* if the host URL should be returned.
- `trusted_hosts` – a list of trusted hosts, see *`host_is_trusted()`* for more information.

`werkzeug.wsgi.get_query_string(`*environ*`)`
Returns the *QUERY_STRING* from the WSGI environment. This also takes care about the WSGI decoding dance on Python 3 environments as a native string. The string returned will be restricted to ASCII characters.

New in version 0.9.

Parameters `environ` – the WSGI environment object to get the query string from.

`werkzeug.wsgi.get_script_name(`*environ*, *charset='utf-8'*, *errors='replace'*`)`
Returns the *SCRIPT_NAME* from the WSGI environment and properly decodes it. This also takes care about the WSGI decoding dance on Python 3 environments. if the *charset* is set to *None* a bytestring is returned.

New in version 0.9.

Parameters

- `environ` – the WSGI environment object to get the path from.
- `charset` – the charset for the path, or *None* if no decoding should be performed.
- `errors` – the decoding error handling.

`werkzeug.wsgi.get_path_info(`*environ*, *charset='utf-8'*, *errors='replace'*`)`
Returns the *PATH_INFO* from the WSGI environment and properly decodes it. This also takes care about the WSGI decoding dance on Python 3 environments. if the *charset* is set to *None* a bytestring is returned.

New in version 0.9.

Parameters

- `environ` – the WSGI environment object to get the path from.
- `charset` – the charset for the path info, or *None* if no decoding should be performed.
- `errors` – the decoding error handling.

`werkzeug.wsgi.pop_path_info(`*environ*, *charset='utf-8'*, *errors='replace'*`)`
Removes and returns the next segment of *PATH_INFO*, pushing it onto *SCRIPT_NAME*. Returns *None* if there is nothing left on *PATH_INFO*.

If the *charset* is set to *None* a bytestring is returned.

If there are empty segments (’/foo//bar) these are ignored but properly pushed to the *SCRIPT_NAME*:

```
>>> env = {'SCRIPT_NAME': '/foo', 'PATH_INFO': '/a/b'}
>>> pop_path_info(env)
'a'
>>> env['SCRIPT_NAME']
'/foo/a'
>>> pop_path_info(env)
'b'
>>> env['SCRIPT_NAME']
'/foo/a/b'
```

New in version 0.5.

Changed in version 0.9: The path is now decoded and a charset and encoding parameter can be provided.

Parameters `environ` – the WSGI environment that is modified.

`werkzeug.wsgi.peek_path_info`(*environ*, *charset=’utf-8’*, *errors=’replace’*)

Returns the next segment on the *PATH_INFO* or *None* if there is none. Works like *pop_path_info()* without modifying the environment:

```
>>> env = {'SCRIPT_NAME': '/foo', 'PATH_INFO': '/a/b'}
>>> peek_path_info(env)
'a'
>>> peek_path_info(env)
'a'
```

If the *charset* is set to *None* a bytestring is returned.

New in version 0.5.

Changed in version 0.9: The path is now decoded and a charset and encoding parameter can be provided.

Parameters `environ` – the WSGI environment that is checked.

`werkzeug.wsgi.extract_path_info`(*environ_or_baseurl*, *path_or_url*, *charset=’utf-8’*, *errors=’replace’*, *collapse_http_schemes=True*)

Extracts the path info from the given URL (or WSGI environment) and path. The path info returned is a unicode string, not a bytestring suitable for a WSGI environment. The URLs might also be IRIs.

If the path info could not be determined, *None* is returned.

Some examples:

```
>>> extract_path_info('http://example.com/app', '/app/hello')
u'/hello'
>>> extract_path_info('http://example.com/app',
...                   'https://example.com/app/hello')
u'/hello'
```

```
>>> extract_path_info('http://example.com/app',
...                   'https://example.com/app/hello',
...                   collapse_http_schemes=False) is None
True
```

Instead of providing a base URL you can also pass a WSGI environment.

New in version 0.6.

Parameters

- `environ_or_baseurl` – a WSGI environment dict, a base URL or base IRI. This is the root of the application.
- `path_or_url` – an absolute path from the server root, a relative path (in which case it's the path info) or a full URL. Also accepts IRIs and unicode parameters.
- `charset` – the charset for byte data in URLs
- `errors` – the error handling on decode
- `collapse_http_schemes` – if set to *False* the algorithm does not assume that http and https on the same server point to the same resource.

`werkzeug.wsgi.host_is_trusted`(*hostname*, *trusted_list*)

Checks if a host is trusted against a list. This also takes care of port normalization.

New in version 0.9.

Parameters

- `hostname` – the hostname to check
- `trusted_list` – a list of hostnames to check against. If a hostname starts with a dot it will match against all subdomains as well.

12.3 Convenience Helpers

`werkzeug.wsgi.responder`(*f*)

Marks a function as responder. Decorate a function with it and it will automatically call the return value as WSGI application.

Example:

```
@responder
def application(environ, start_response):
    return Response('Hello World!')
```

`werkzeug.testapp.test_app`(*environ*, *start_response*)

Simple test application that dumps the environment. You can use it to check if Werkzeug is working properly:

```
>>> from werkzeug.serving import run_simple
>>> from werkzeug.testapp import test_app
>>> run_simple('localhost', 3000, test_app)
 * Running on http://localhost:3000/
```

The application displays important information from the WSGI environment, the Python interpreter and the installed libraries.

CHAPTER

THIRTEEN

HTTP UTILITIES

Werkzeug provides a couple of functions to parse and generate HTTP headers that are useful when implementing WSGI middlewares or whenever you are operating on a lower level layer. All this functionality is also exposed from request and response objects.

13.1 Date Functions

The following functions simplify working with times in an HTTP context. Werkzeug uses offset-naive `datetime` objects internally that store the time in UTC. If you're working with timezones in your application make sure to replace the tzinfo attribute with a UTC timezone information before processing the values.

`werkzeug.http.cookie_date`(*expires=None*)

Formats the time to ensure compatibility with Netscape's cookie standard.

Accepts a floating point number expressed in seconds since the epoch in, a datetime object or a timetuple. All times in UTC. The *`parse_date()`* function can be used to parse such a date.

Outputs a string in the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`.

Parameters `expires` – If provided that date is used, otherwise the current.

`werkzeug.http.http_date`(*timestamp=None*)

Formats the time to match the RFC1123 date format.

Accepts a floating point number expressed in seconds since the epoch in, a datetime object or a timetuple. All times in UTC. The *`parse_date()`* function can be used to parse such a date.

Outputs a string in the format `Wdy, DD Mon YYYY HH:MM:SS GMT`.

Parameters `timestamp` – If provided that date is used, otherwise the current.

`werkzeug.http.parse_date`(*value*)

Parse one of the following date formats into a datetime object:

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime() format
```

If parsing fails the return value is *None*.

Parameters `value` – a string with a supported date format.

Returns a `datetime.datetime` object.

13.2 Header Parsing

The following functions can be used to parse incoming HTTP headers. Because Python does not provide data structures with the semantics required by **RFC 2616**, Werkzeug implements some custom data structures that are *documented separately*.

`werkzeug.http.parse_options_header(value)`

Parse a `Content-Type` like header into a tuple with the content type and the options:

```
>>> parse_options_header('text/html; charset=utf8')
('text/html', {'charset': 'utf8'})
```

This should not be used to parse `Cache-Control` like headers that use a slightly different format. For these headers use the *`parse_dict_header()`* function.

New in version 0.5.

Parameters `value` – the header to parse.

Returns (str, options)

`werkzeug.http.parse_set_header(value, on_update=None)`

Parse a set-like header and return a *`HeaderSet`* object:

```
>>> hs = parse_set_header('token, "quoted value"')
```

The return value is an object that treats the items case-insensitively and keeps the order of the items:

```
>>> 'TOKEN' in hs
True
>>> hs.index('quoted value')
1
>>> hs
HeaderSet(['token', 'quoted value'])
```

To create a header from the `HeaderSet` again, use the *`dump_header()`* function.

Parameters

- `value` – a set header to be parsed.

- `on_update` – an optional callable that is called every time a value on the *HeaderSet* object is changed.

Returns a *HeaderSet*

`werkzeug.http.parse_list_header(value)`

Parse lists as described by RFC 2068 Section 2.

In particular, parse comma-separated lists where the elements of the list may include quoted-strings. A quoted-string could contain a comma. A non-quoted string could have quotes in the middle. Quotes are removed automatically after parsing.

It basically works like *parse_set_header()* just that items may appear multiple times and case sensitivity is preserved.

The return value is a standard `list`:

```
>>> parse_list_header('token, "quoted value"')
['token', 'quoted value']
```

To create a header from the `list` again, use the *dump_header()* function.

Parameters `value` – a string with a list header.

Returns `list`

`werkzeug.http.parse_dict_header(value, cls=<class 'dict'>)`

Parse lists of key, value pairs as described by RFC 2068 Section 2 and convert them into a python dict (or any other mapping object created from the type with a dict like interface provided by the *cls* arugment):

```
>>> d = parse_dict_header('foo="is a fish", bar="as well"')
>>> type(d) is dict
True
>>> sorted(d.items())
[('bar', 'as well'), ('foo', 'is a fish')]
```

If there is no value for a key it will be *None*:

```
>>> parse_dict_header('key_without_value')
{'key_without_value': None}
```

To create a header from the `dict` again, use the *dump_header()* function.

Changed in version 0.9: Added support for *cls* argument.

Parameters

- `value` – a string with a dict header.
- `cls` – callable to use for storage of parsed results.

Returns an instance of *cls*

`werkzeug.http.parse_accept_header(`*value*[, *class*]`)`

Parses an HTTP Accept-* header. This does not implement a complete valid algorithm but one that supports at least value and quality extraction.

Returns a new `Accept` object (basically a list of `(value, quality)` tuples sorted by the quality with some additional accessor methods).

The second parameter can be a subclass of `Accept` that is created with the parsed values and returned.

Parameters

- `value` – the accept header string to be parsed.
- `cls` – the wrapper class for the return value (can be `Accept` or a subclass thereof)

Returns an instance of *cls*.

`werkzeug.http.parse_cache_control_header(`*value*, *on_update=None*, *cls=None*`)`

Parse a cache control header. The RFC differs between response and request cache control, this method does not. It's your responsibility to not use the wrong control statements.

New in version 0.5: The *cls* was added. If not specified an immutable *`RequestCacheControl`* is returned.

Parameters

- `value` – a cache control header to be parsed.
- `on_update` – an optional callable that is called every time a value on the `CacheControl` object is changed.
- `cls` – the class for the returned object. By default *`RequestCacheControl`* is used.

Returns a *cls* object.

`werkzeug.http.parse_authorization_header(`*value*`)`

Parse an HTTP basic/digest authorization header transmitted by the web browser. The return value is either *None* if the header was invalid or not given, otherwise an *`Authorization`* object.

Parameters `value` – the authorization header to parse.

Returns a *`Authorization`* object or *None*.

`werkzeug.http.parse_www_authenticate_header(`*value*, *on_update=None*`)`

Parse an HTTP WWW-Authenticate header into a *`WWWAuthenticate`* object.

Parameters

- `value` – a WWW-Authenticate header to parse.
- `on_update` – an optional callable that is called every time a value on the *`WWWAuthenticate`* object is changed.

Returns a *WWWAuthenticate* object.

werkzeug.http.parse_if_range_header(*value*)
Parses an if-range header which can be an etag or a date. Returns a *IfRange* object.

New in version 0.7.

werkzeug.http.parse_range_header(*value*, *make_inclusive=True*)
Parses a range header into a *Range* object. If the header is missing or malformed *None* is returned. *ranges* is a list of `(start, stop)` tuples where the ranges are non-inclusive.

New in version 0.7.

werkzeug.http.parse_content_range_header(*value*, *on_update=None*)
Parses a range header into a *ContentRange* object or *None* if parsing is not possible.

New in version 0.7.

Parameters

- `value` – a content range header to be parsed.
- `on_update` – an optional callable that is called every time a value on the *ContentRange* object is changed.

13.3 Header Utilities

The following utilities operate on HTTP headers well but do not parse them. They are useful if you're dealing with conditional responses or if you want to proxy arbitrary requests but want to remove WSGI-unsupported hop-by-hop headers. Also there is a function to create HTTP header strings from the parsed data.

werkzeug.http.is_entity_header(*header*)
Check if a header is an entity header.

New in version 0.5.

Parameters `header` – the header to test.

Returns *True* if it's an entity header, *False* otherwise.

werkzeug.http.is_hop_by_hop_header(*header*)
Check if a header is an HTTP/1.1 "Hop-by-Hop" header.

New in version 0.5.

Parameters `header` – the header to test.

Returns *True* if it's an entity header, *False* otherwise.

werkzeug.http.remove_entity_headers(*headers*, *allowed=('expires', 'content-location')*)
Remove all entity headers from a list or `Headers` object. This operation works

in-place. *Expires* and *Content-Location* headers are by default not removed. The reason for this is RFC 2616 section 10.3.5 which specifies some entity headers that should be sent.

Changed in version 0.5: added *allowed* parameter.

Parameters

- `headers` – a list or `Headers` object.
- `allowed` – a list of headers that should still be allowed even though they are entity headers.

`werkzeug.http.remove_hop_by_hop_headers(`*headers*`)`
Remove all HTTP/1.1 "Hop-by-Hop" headers from a list or `Headers` object. This operation works in-place.

New in version 0.5.

Parameters `headers` – a list or `Headers` object.

`werkzeug.http.is_byte_range_valid(`*start*, *stop*, *length*`)`
Checks if a given byte content range is valid for the given length.

New in version 0.7.

`werkzeug.http.quote_header_value(`*value*, *extra_chars=''*, *allow_token=True*`)`
Quote a header value if necessary.

New in version 0.5.

Parameters

- `value` – the value to quote.
- `extra_chars` – a list of extra characters to skip quoting.
- `allow_token` – if this is enabled token values are returned unchanged.

`werkzeug.http.unquote_header_value(`*value*, *is_filename=False*`)`
Unquotes a header value. (Reversal of *`quote_header_value()`*). This does not use the real unquoting but what browsers are actually using for quoting.

New in version 0.5.

Parameters `value` – the header value to unquote.

`werkzeug.http.dump_header(`*iterable*, *allow_token=True*`)`
Dump an HTTP header again. This is the reversal of *`parse_list_header()`*, *`parse_set_header()`* and *`parse_dict_header()`*. This also quotes strings that include an equals sign unless you pass it as dict of key, value pairs.

```
>>> dump_header({'foo': 'bar baz'})
'foo="bar baz"'
>>> dump_header(('foo', 'bar baz'))
'foo, "bar baz"'
```

Parameters

- `iterable` – the iterable or dict of values to quote.
- `allow_token` – if set to *False* tokens as values are disallowed. See *quote_header_value()* for more details.

13.4 Cookies

`werkzeug.http.parse_cookie`(*header*, *charset='utf-8'*, *errors='replace'*, *cls=None*)

Parse a cookie. Either from a string or WSGI environ.

Per default encoding errors are ignored. If you want a different behavior you can set *errors* to `'replace'` or `'strict'`. In strict mode a `HTTPUnicodeError` is raised.

Changed in version 0.5: This function now returns a `TypeConversionDict` instead of a regular dict. The *cls* parameter was added.

Parameters

- `header` – the header to be used to parse the cookie. Alternatively this can be a WSGI environment.
- `charset` – the charset for the cookie values.
- `errors` – the error behavior for the charset decoding.
- `cls` – an optional dict class to use. If this is not specified or *None* the default `TypeConversionDict` is used.

`werkzeug.http.dump_cookie`(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *charset='utf-8'*, *sync_expires=True*)

Creates a new Set-Cookie header without the `Set-Cookie` prefix The parameters are the same as in the cookie Morsel object in the Python standard library but it accepts unicode data, too.

On Python 3 the return value of this function will be a unicode string, on Python 2 it will be a native string. In both cases the return value is usually restricted to ascii as the vast majority of values are properly escaped, but that is no guarantee. If a unicode string is returned it's tunneled through latin1 as required by PEP 3333.

The return value is not ASCII safe if the key contains unicode characters. This is technically against the specification but happens in the wild. It's strongly recommended to not use non-ASCII values for the keys.

Parameters

- `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. Additionally *timedelta* objects are accepted, too.
- `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` – Use this 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` – 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.
- `charset` – the encoding for unicode values.
- `sync_expires` – automatically set expires if max_age is defined but expires not.

13.5 Conditional Response Helpers

For conditional responses the following functions might be useful:

`werkzeug.http.parse_etags(`*value*`)`
Parse an etag header.

Parameters `value` – the tag header to parse

Returns an *`ETags`* object.

`werkzeug.http.quote_etag(`*etag*, *weak=False*`)`
Quote an etag.

Parameters

- `etag` – the etag to quote.
- `weak` – set to *True* to tag it "weak".

`werkzeug.http.unquote_etag(`*etag*`)`
Unquote a single etag:

```
>>> unquote_etag('w/"bar"')
('bar', True)
>>> unquote_etag('"bar"')
('bar', False)
```

Parameters `etag` – the etag identifier to unquote.

Returns a `(etag, weak)` tuple.

`werkzeug.http.generate_etag(`*data*`)`
Generate an etag for some data.

werkzeug.http.is_resource_modified(*environ*, *etag=None*, *data=None*, *last_modified=None*)

Convenience method for conditional requests.

Parameters

- `environ` – the WSGI environment of the request to be checked.
- `etag` – the etag for the response for comparison.
- `data` – or alternatively the data of the response to automatically generate an etag using *generate_etag()*.
- `last_modified` – an optional date of the last modification.

Returns *True* if the resource was modified, otherwise *False*.

13.6 Constants

werkzeug.http.HTTP_STATUS_CODES

A dict of status code -> default status message pairs. This is used by the wrappers and other places where an integer status code is expanded to a string throughout Werkzeug.

13.7 Form Data Parsing

Werkzeug provides the form parsing functions separately from the request object so that you can access form data from a plain WSGI environment.

The following formats are currently supported by the form data parser:

- *application/x-www-form-urlencoded*
- *multipart/form-data*

Nested multipart is not currently supported (Werkzeug 0.9), but it isn't used by any of the modern web browsers.

Usage example:

```
>>> from cStringIO import StringIO
>>> data = '--foo\r\nContent-Disposition: form-data; name="test"\r\n' \
... '\r\nHello World!\r\n--foo--'
>>> environ = {'wsgi.input': StringIO(data), 'CONTENT_LENGTH': str(len(data)),
...            'CONTENT_TYPE': 'multipart/form-data; boundary=foo',
...            'REQUEST_METHOD': 'POST'}
>>> stream, form, files = parse_form_data(environ)
>>> stream.read()
''
>>> form['test']
u'Hello World!'
```

```
>>> not files
True
```

Normally the WSGI environment is provided by the WSGI gateway with the incoming data as part of it. If you want to generate such fake-WSGI environments for unittesting you might want to use the `create_environ()` function or the `EnvironBuilder` instead.

class `werkzeug.formparser.FormDataParser`(*stream_factory=None*, *charset='utf-8'*, *errors='replace'*, *max_form_memory_size=None*, *max_content_length=None*, *cls=None*, *silent=True*)

This class implements parsing of form data for Werkzeug. By itself it can parse multipart and url encoded form data. It can be subclassed and extended but for most mimetypes it is a better idea to use the untouched stream and expose it as separate attributes on a request object.

New in version 0.8.

Parameters

- `stream_factory` – An optional callable that returns a new read and writeable file descriptor. This callable works the same as `_get_file_stream()`.
- `charset` – The character set for URL and url encoded form data.
- `errors` – The encoding error behavior.
- `max_form_memory_size` – the maximum number of bytes to be accepted for in-memory stored form data. If the data exceeds the value specified an `RequestEntityTooLarge` exception is raised.
- `max_content_length` – If this is provided and the transmitted data is longer than this value an `RequestEntityTooLarge` exception is raised.
- `cls` – an optional dict class to use. If this is not specified or *None* the default `MultiDict` is used.
- `silent` – If set to False parsing errors will not be caught.

`werkzeug.formparser.parse_form_data`(*environ*, *stream_factory=None*, *charset='utf-8'*, *errors='replace'*, *max_form_memory_size=None*, *max_content_length=None*, *cls=None*, *silent=True*)

Parse the form data in the environ and return it as tuple in the form `(stream, form, files)`. You should only call this method if the transport method is *POST*, *PUT*, or *PATCH*.

If the mimetype of the data transmitted is *multipart/form-data* the files multidict will be filled with *FileStorage* objects. If the mimetype is unknown the input stream is wrapped and returned as first argument, else the stream is empty.

This is a shortcut for the common usage of *FormDataParser*.

Have a look at *Dealing with Request Data* for more details.

New in version 0.5: The *max_form_memory_size*, *max_content_length* and *cls* parameters were added.

New in version 0.5.1: The optional *silent* flag was added.

Parameters

- `environ` – the WSGI environment to be used for parsing.
- `stream_factory` – An optional callable that returns a new read and writeable file descriptor. This callable works the same as `_get_file_stream()`.
- `charset` – The character set for URL and url encoded form data.
- `errors` – The encoding error behavior.
- `max_form_memory_size` – the maximum number of bytes to be accepted for in-memory stored form data. If the data exceeds the value specified an `RequestEntityTooLarge` exception is raised.
- `max_content_length` – If this is provided and the transmitted data is longer than this value an `RequestEntityTooLarge` exception is raised.
- `cls` – an optional dict class to use. If this is not specified or *None* the default `MultiDict` is used.
- `silent` – If set to False parsing errors will not be caught.

Returns A tuple in the form `(stream, form, files)`.

`werkzeug.formparser.parse_multipart_headers(`*iterable*`)`

Parses multipart headers from an iterable that yields lines (including the trailing newline symbol). The iterable has to be newline terminated.

The iterable will stop at the line where the headers ended so it can be further consumed.

Parameters `iterable` – iterable of strings that are newline terminated

CHAPTER

FOURTEEN

DATA STRUCTURES

Werkzeug provides some subclasses of common Python objects to extend them with additional features. Some of them are used to make them immutable, others are used to change some semantics to better work with HTTP.

14.1 General Purpose

Changed in version 0.6: The general purpose classes are now pickleable in each protocol as long as the contained objects are pickleable. This means that the *FileMultiDict* won't be pickleable as soon as it contains a file.

class `werkzeug.datastructures.TypeConversionDict`

Works like a regular dict but the *get()* method can perform type conversions. *MultiDict* and *CombinedMultiDict* are subclasses of this class and provide the same feature.

New in version 0.5.

`get`(*key*, *default=None*, *type=None*)

Return the default value if the requested data doesn't exist. If *type* is provided and is a callable it should convert the value, return it or raise a `ValueError` if that is not possible. In this case the function will return the default as if the value was not found:

```
>>> d = TypeConversionDict(foo='42', bar='blub')
>>> d.get('foo', type=int)
42
>>> d.get('bar', -1, type=int)
-1
```

Parameters

- `key` – The key to be looked up.
- `default` – The default value to be returned if the key can't be looked up. If not further specified *None* is returned.

- `type` – A callable that is used to cast the value in the *MultiDict*. If a `ValueError` is raised by this callable the default value is returned.

class `werkzeug.datastructures.ImmutableTypeConversionDict`
Works like a *TypeConversionDict* but does not support modifications.

New in version 0.5.

`copy()`
Return a shallow mutable copy of this object. Keep in mind that the standard library's *copy()* function is a no-op for this class like for any other python immutable type (eg: `tuple`).

class `werkzeug.datastructures.MultiDict(`*mapping=None*`)`
A *MultiDict* is a dictionary subclass customized to deal with multiple values for the same key which is for example used by the parsing functions in the wrappers. This is necessary because some HTML form elements pass multiple values for the same key.

MultiDict implements all standard dictionary methods. Internally, it saves all values for a key as a list, but the standard dict access methods will only return the first value for a key. If you want to gain access to the other values, too, you have to use the *list* methods as explained below.

Basic Usage:

```
>>> d = MultiDict([('a', 'b'), ('a', 'c')])
>>> d
MultiDict([('a', 'b'), ('a', 'c')])
>>> d['a']
'b'
>>> d.getlist('a')
['b', 'c']
>>> 'a' in d
True
```

It behaves like a normal dict thus all dict functions will only return the first value when multiple values for one key are found.

From Werkzeug 0.3 onwards, the *KeyError* raised by this class is also a subclass of the `BadRequest` HTTP exception and will render a page for a `400 BAD REQUEST` if caught in a catch-all for HTTP exceptions.

A *MultiDict* can be constructed from an iterable of `(key, value)` tuples, a dict, a *MultiDict* or from Werkzeug 0.2 onwards some keyword parameters.

Parameters `mapping` – the initial value for the *MultiDict*. Either a regular dict, an iterable of `(key, value)` tuples or *None*.

`add(`*key*, *value*`)`
Adds a new value for the key.

New in version 0.6.

Parameters

- `key` – the key for the value.
- `value` – the value to add.

`clear()` → None. Remove all items from D.

`copy()`
Return a shallow copy of this object.

`deepcopy(memo=None)`
Return a deep copy of this object.

`fromkeys()`
Returns a new dict with keys from iterable and values equal to value.

`get(key, default=None, type=None)`
Return the default value if the requested data doesn't exist. If *type* is provided and is a callable it should convert the value, return it or raise a `ValueError` if that is not possible. In this case the function will return the default as if the value was not found:

```
>>> d = TypeConversionDict(foo='42', bar='blub')
>>> d.get('foo', type=int)
42
>>> d.get('bar', -1, type=int)
-1
```

Parameters

- `key` – The key to be looked up.
- `default` – The default value to be returned if the key can't be looked up. If not further specified *None* is returned.
- `type` – A callable that is used to cast the value in the *MultiDict*. If a `ValueError` is raised by this callable the default value is returned.

`getlist(key, type=None)`
Return the list of items for a given key. If that key is not in the *MultiDict*, the return value will be an empty list. Just as *get* *getlist* accepts a *type* parameter. All items will be converted with the callable defined there.

Parameters

- `key` – The key to be looked up.
- `type` – A callable that is used to cast the value in the *MultiDict*. If a `ValueError` is raised by this callable the value will be removed from the list.

Returns a `list` of all the values for the key.

`items`(*multi=False*)
Return an iterator of `(key, value)` pairs.

Parameters `multi` – If set to *True* the iterator returned will have a pair for each value of each key. Otherwise it will only contain pairs for the first value of each key.

`lists`()
Return a list of `(key, values)` pairs, where values is the list of all values associated with the key.

`listvalues`()
Return an iterator of all values associated with a key. Zipping `keys()` and this is the same as calling *`lists()`*:

```
>>> d = MultiDict({"foo": [1, 2, 3]})
>>> zip(d.keys(), d.listvalues()) == d.lists()
True
```

`pop`(*key*, *default=no value*)
Pop the first item for a list on the dict. Afterwards the key is removed from the dict, so additional values are discarded:

```
>>> d = MultiDict({"foo": [1, 2, 3]})
>>> d.pop("foo")
1
>>> "foo" in d
False
```

Parameters

- `key` – the key to pop.
- `default` – if provided the value to return if the key was not in the dictionary.

`popitem`()
Pop an item from the dict.

`popitemlist`()
Pop a `(key, list)` tuple from the dict.

`poplist`(*key*)
Pop the list for a key from the dict. If the key is not in the dict an empty list is returned.

Changed in version 0.5: If the key does no longer exist a list is returned instead of raising an error.

`setdefault`(*key*, *default=None*)
Returns the value for the key if it is in the dict, otherwise it returns *default* and sets that value for *key*.

Parameters

- `key` – The key to be looked up.
- `default` – The default value to be returned if the key is not in the dict. If not further specified it's *None*.

`setlist`(*key*, *new_list*)

Remove the old values for a key and add new ones. Note that the list you pass the values in will be shallow-copied before it is inserted in the dictionary.

```
>>> d = MultiDict()
>>> d.setlist('foo', ['1', '2'])
>>> d['foo']
'1'
>>> d.getlist('foo')
['1', '2']
```

Parameters

- `key` – The key for which the values are set.
- `new_list` – An iterable with the new values for the key. Old values are removed first.

`setlistdefault`(*key*, *default_list=None*)

Like *setdefault* but sets multiple values. The list returned is not a copy, but the list that is actually used internally. This means that you can put new values into the dict by appending items to the list:

```
>>> d = MultiDict({"foo": 1})
>>> d.setlistdefault("foo").extend([2, 3])
>>> d.getlist("foo")
[1, 2, 3]
```

Parameters

- `key` – The key to be looked up.
- `default` – An iterable of default values. It is either copied (in case it was a list) or converted into a list before returned.

Returns a `list`

`to_dict`(*flat=True*)

Return the contents as regular dict. If *flat* is *True* the returned dict will only have the first item present, if *flat* is *False* all values will be returned as lists.

Parameters `flat` – If set to *False* the dict returned will have lists with all the values in it. Otherwise it will only contain the first value for each key.

Returns a `dict`

update(*other_dict*)
update() extends rather than replaces existing key lists.

values()
Returns an iterator of the first value on every key's value list.

class werkzeug.datastructures.OrderedMultiDict(*mapping=None*)
Works like a regular *MultiDict* but preserves the order of the fields. To convert the ordered multi dict into a list you can use the items() method and pass it multi=True.

In general an *OrderedMultiDict* is an order of magnitude slower than a *MultiDict*.

---

note

Due to a limitation in Python you cannot convert an ordered multi dict into a regular dict by using dict(multidict). Instead you have to use the to_dict() method, otherwise the internal bucket objects are exposed.

---

class werkzeug.datastructures.ImmutableMultiDict(*mapping=None*)
An immutable *MultiDict*.

New in version 0.5.

copy()
Return a shallow mutable copy of this object. Keep in mind that the standard library's *copy()* function is a no-op for this class like for any other python immutable type (eg: tuple).

class werkzeug.datastructures.ImmutableOrderedMultiDict(*mapping=None*)
An immutable *OrderedMultiDict*.

New in version 0.6.

copy()
Return a shallow mutable copy of this object. Keep in mind that the standard library's *copy()* function is a no-op for this class like for any other python immutable type (eg: tuple).

class werkzeug.datastructures.CombinedMultiDict(*dicts=None*)
A read only *MultiDict* that you can pass multiple *MultiDict* instances as sequence and it will combine the return values of all wrapped dicts:

```
>>> from werkzeug.datastructures import CombinedMultiDict, MultiDict
>>> post = MultiDict([('foo', 'bar')])
>>> get = MultiDict([('blub', 'blah')])
>>> combined = CombinedMultiDict([get, post])
>>> combined['foo']
'bar'
>>> combined['blub']
'blah'
```

This works for all read operations and will raise a *TypeError* for methods that usually change data which isn't possible.

From Werkzeug 0.3 onwards, the *KeyError* raised by this class is also a subclass of the `BadRequest` HTTP exception and will render a page for a `400 BAD REQUEST` if caught in a catch-all for HTTP exceptions.

class `werkzeug.datastructures.ImmutableDict`
An immutable `dict`.

New in version 0.5.

`copy()`
Return a shallow mutable copy of this object. Keep in mind that the standard library's *`copy()`* function is a no-op for this class like for any other python immutable type (eg: `tuple`).

class `werkzeug.datastructures.ImmutableList`
An immutable `list`.

New in version 0.5.

Private

class `werkzeug.datastructures.FileMultiDict(`*mapping=None*`)`
A special *`MultiDict`* that has convenience methods to add files to it. This is used for `EnvironBuilder` and generally useful for unittesting.

New in version 0.5.

`add_file(`*name, file, filename=None, content_type=None*`)`
Adds a new file to the dict. *file* can be a file name or a `file`-like or a *`FileStorage`* object.

Parameters

- `name` – the name of the field.
- `file` – a filename or `file`-like object
- `filename` – an optional filename
- `content_type` – an optional content type

14.2 HTTP Related

class `werkzeug.datastructures.Headers(`[*defaults*]`)`
An object that stores some headers. It has a dict-like interface but is ordered and can store the same keys multiple times.

This data structure is useful if you want a nicer way to handle WSGI headers which are stored as tuples in a list.

From Werkzeug 0.3 onwards, the KeyError raised by this class is also a subclass of the BadRequest HTTP exception and will render a page for a 400 BAD REQUEST if caught in a catch-all for HTTP exceptions.

Headers is mostly compatible with the Python wsgiref.headers.Headers class, with the exception of __getitem__. wsgiref will return *None* for headers['missing'], whereas *Headers* will raise a KeyError.

To create a new *Headers* object pass it a list or dict of headers which are used as default values. This does not reuse the list passed to the constructor for internal usage.

Parameters defaults – The list of default values for the *Headers*.

Changed in version 0.9: This data structure now stores unicode values similar to how the multi dicts do it. The main difference is that bytes can be set as well which will automatically be latin1 decoded.

Changed in version 0.9: The linked() function was removed without replacement as it was an API that does not support the changes to the encoding model.

add(*_key*, *_value*, ***kw*)

Add a new header tuple to the list.

Keyword arguments can specify additional parameters for the header value, with underscores converted to dashes:

```
>>> d = Headers()
>>> d.add('Content-Type', 'text/plain')
>>> d.add('Content-Disposition', 'attachment', filename='foo.png')
```

The keyword argument dumping uses dump_options_header() behind the scenes.

New in version 0.4.1: keyword arguments were added for wsgiref compatibility.

add_header(*_key*, *_value*, ***_kw*)

Add a new header tuple to the list.

An alias for *add()* for compatibility with the wsgiref add_header() method.

clear()

Clears all headers.

extend(*iterable*)

Extend the headers with a dict or an iterable yielding keys and values.

get(*key*, *default=None*, *type=None*, *as_bytes=False*)

Return the default value if the requested data doesn't exist. If *type* is provided and is a callable it should convert the value, return it or raise a ValueError if that is not possible. In this case the function will return the default as if the value was not found:

```
>>> d = Headers([('Content-Length', '42')])
>>> d.get('Content-Length', type=int)
42
```

If a headers object is bound you must not add unicode strings because no encoding takes place.

New in version 0.9: Added support for *as_bytes*.

Parameters

- `key` – The key to be looked up.
- `default` – The default value to be returned if the key can't be looked up. If not further specified *None* is returned.
- `type` – A callable that is used to cast the value in the *Headers*. If a `ValueError` is raised by this callable the default value is returned.
- `as_bytes` – return bytes instead of unicode strings.

`get_all(name)`
Return a list of all the values for the named field.

This method is compatible with the `wsgiref` `get_all()` method.

`getlist(key, type=None, as_bytes=False)`
Return the list of items for a given key. If that key is not in the *Headers*, the return value will be an empty list. Just as *get()* *getlist()* accepts a *type* parameter. All items will be converted with the callable defined there.

New in version 0.9: Added support for *as_bytes*.

Parameters

- `key` – The key to be looked up.
- `type` – A callable that is used to cast the value in the *Headers*. If a `ValueError` is raised by this callable the value will be removed from the list.
- `as_bytes` – return bytes instead of unicode strings.

Returns a `list` of all the values for the key.

`has_key(key)`
Check if a key is present.

`pop(key=None, default=no value)`
Removes and returns a key or index.

Parameters `key` – The key to be popped. If this is an integer the item at that position is removed, if it's a string the value for that key is. If the key is omitted or *None* the last item is removed.

Returns an item.

`popitem()`
Removes a key or index and returns a (key, value) item.

`remove(`*key*`)`
Remove a key.

Parameters `key` – The key to be removed.

`set(`*_key*, *_value*, ***kw*`)`
Remove all header tuples for *key* and add a new one. The newly added key either appears at the end of the list if there was no entry or replaces the first one.

Keyword arguments can specify additional parameters for the header value, with underscores converted to dashes. See *`add()`* for more information.

Changed in version 0.6.1: *`set()`* now accepts the same arguments as *`add()`*.

Parameters

- `key` – The key to be inserted.
- `value` – The value to be inserted.

`setdefault(`*key*, *value*`)`
Returns the value for the key if it is in the dict, otherwise it returns *default* and sets that value for *key*.

Parameters

- `key` – The key to be looked up.
- `default` – The default value to be returned if the key is not in the dict. If not further specified it's *None*.

`to_list(`*charset='iso-8859-1'*`)`
Convert the headers into a list suitable for WSGI.

`to_wsgi_list()`
Convert the headers into a list suitable for WSGI.

The values are byte strings in Python 2 converted to latin1 and unicode strings in Python 3 for the WSGI server to encode.

Returns list

class `werkzeug.datastructures.EnvironHeaders(`*environ*`)`
Read only version of the headers from a WSGI environment. This provides the same interface as *Headers* and is constructed from a WSGI environment.

From Werkzeug 0.3 onwards, the *KeyError* raised by this class is also a subclass of the `BadRequest` HTTP exception and will render a page for a `400 BAD REQUEST` if caught in a catch-all for HTTP exceptions.

class `werkzeug.datastructures.HeaderSet(`*headers=None*, *on_update=None*`)`
Similar to the *`ETags`* class this implements a set-like structure. Unlike *`ETags`* this

is case insensitive and used for vary, allow, and content-language headers.

If not constructed using the `parse_set_header()` function the instantiation works like this:

```
>>> hs = HeaderSet(['foo', 'bar', 'baz'])
>>> hs
HeaderSet(['foo', 'bar', 'baz'])
```

`add(`*header*`)`
: Add a new header to the set.

`as_set(`*preserve_casing=False*`)`
: Return the set as real python set type. When calling this, all the items are converted to lowercase and the ordering is lost.

  Parameters `preserve_casing` – if set to *True* the items in the set returned will have the original case like in the *HeaderSet*, otherwise they will be lowercase.

`clear()`
: Clear the set.

`discard(`*header*`)`
: Like *remove()* but ignores errors.

  Parameters `header` – the header to be discarded.

`find(`*header*`)`
: Return the index of the header in the set or return -1 if not found.

  Parameters `header` – the header to be looked up.

`index(`*header*`)`
: Return the index of the header in the set or raise an `IndexError`.

  Parameters `header` – the header to be looked up.

`remove(`*header*`)`
: Remove a header from the set. This raises an `KeyError` if the header is not in the set.

  Changed in version 0.5: In older versions a `IndexError` was raised instead of a `KeyError` if the object was missing.

  Parameters `header` – the header to be removed.

`to_header()`
: Convert the header set into an HTTP header string.

`update(`*iterable*`)`
: Add all the headers from the iterable to the set.

  Parameters `iterable` – updates the set with the items from the iterable.

class `werkzeug.datastructures.Accept(`*values=()*`)`

An *Accept* object is just a list subclass for lists of `(value, quality)` tuples. It is automatically sorted by quality.

All *Accept* objects work similar to a list but provide extra functionality for working with the data. Containment checks are normalized to the rules of that header:

```
>>> a = CharsetAccept([('ISO-8859-1', 1), ('utf-8', 0.7)])
>>> a.best
'ISO-8859-1'
>>> 'iso-8859-1' in a
True
>>> 'UTF8' in a
True
>>> 'utf7' in a
False
```

To get the quality for an item you can use normal item lookup:

```
>>> print a['utf-8']
0.7
>>> a['utf7']
0
```

Changed in version 0.5: *Accept* objects are forced immutable now.

`best`

The best match as value.

`best_match(`*matches*, *default=None*`)`

Returns the best match from a list of possible matches based on the quality of the client. If two items have the same quality, the one is returned that comes first.

Parameters

- `matches` – a list of matches to check for
- `default` – the value that is returned if none match

`find(`*key*`)`

Get the position of an entry or return -1.

Parameters `key` – The key to be looked up.

`index(`*key*`)`

Get the position of an entry or raise `ValueError`.

Parameters `key` – The key to be looked up.

Changed in version 0.5: This used to raise `IndexError`, which was inconsistent with the list API.

`quality(`*key*`)`

Returns the quality of the key.

New in version 0.6: In previous versions you had to use the item-lookup syntax (eg: `obj[key]` instead of `obj.quality(key)`)

`to_header()`
Convert the header set into an HTTP header string.

`values()`
Iterate over all values.

class `werkzeug.datastructures.MIMEAccept(`*values=()*`)`
Like *`Accept`* but with special methods and behavior for mimetypes.

`accept_html`
True if this object accepts HTML.

`accept_json`
True if this object accepts JSON.

`accept_xhtml`
True if this object accepts XHTML.

class `werkzeug.datastructures.CharsetAccept(`*values=()*`)`
Like *`Accept`* but with normalization for charsets.

class `werkzeug.datastructures.LanguageAccept(`*values=()*`)`
Like *`Accept`* but with normalization for languages.

class `werkzeug.datastructures.RequestCacheControl(`*values=(), on_update=None*`)`
A cache control for requests. This is immutable and gives access to all the request-relevant cache control headers.

To get a header of the *`RequestCacheControl`* object again you can convert the object into a string or call the `to_header()` method. If you plan to subclass it and add your own items have a look at the sourcecode for that class.

New in version 0.5: In previous versions a *CacheControl* class existed that was used both for request and response.

`no_cache`
accessor for 'no-cache'

`no_store`
accessor for 'no-store'

`max_age`
accessor for 'max-age'

`no_transform`
accessor for 'no-transform'

`max_stale`
accessor for 'max-stale'

`min_fresh`
accessor for 'min-fresh'

`no_transform`
accessor for 'no-transform'

`only_if_cached`
accessor for 'only-if-cached'

class `werkzeug.datastructures.ResponseCacheControl(`*values=()*, *on_update=None*`)`
A cache control for responses. Unlike *`RequestCacheControl`* this is mutable and gives access to response-relevant cache control headers.

To get a header of the *`ResponseCacheControl`* object again you can convert the object into a string or call the `to_header()` method. If you plan to subclass it and add your own items have a look at the sourcecode for that class.

New in version 0.5: In previous versions a *CacheControl* class existed that was used both for request and response.

`no_cache`
accessor for 'no-cache'

`no_store`
accessor for 'no-store'

`max_age`
accessor for 'max-age'

`no_transform`
accessor for 'no-transform'

`must_revalidate`
accessor for 'must-revalidate'

`private`
accessor for 'private'

`proxy_revalidate`
accessor for 'proxy-revalidate'

`public`
accessor for 'public'

`s_maxage`
accessor for 's-maxage'

class `werkzeug.datastructures.ETags(`*strong_etags=None*, *weak_etags=None*, *star_tag=False*`)`
A set that can be used to check if one etag is present in a collection of etags.

`as_set(`*include_weak=False*`)`
Convert the *ETags* object into a python set. Per default all the weak etags are not part of this set.

`contains(`*etag*`)`
Check if an etag is part of the set ignoring weak tags. It is also possible to use the `in` operator.

`contains_raw(`*etag*`)`
: When passed a quoted tag it will check if this tag is part of the set. If the tag is weak it is checked against weak and strong tags, otherwise strong only.

`contains_weak(`*etag*`)`
: Check if an etag is part of the set including weak and strong tags.

`is_weak(`*etag*`)`
: Check if an etag is weak.

`to_header()`
: Convert the etags set into a HTTP header string.

class `werkzeug.datastructures.Authorization(`*auth_type*, *data=None*`)`
Represents an *Authorization* header sent by the client. You should not create this kind of object yourself but use it when it's returned by the *parse_authorization_header* function.

This object is a dict subclass and can be altered by setting dict items but it should be considered immutable as it's returned by the client and not meant for modifications.

Changed in version 0.5: This object became immutable.

`cnonce`
: If the server sent a qop-header in the `WWW-Authenticate` header, the client has to provide this value for HTTP digest auth. See the RFC for more details.

`nc`
: The nonce count value transmitted by clients if a qop-header is also transmitted. HTTP digest auth only.

`nonce`
: The nonce the server sent for digest auth, sent back by the client. A nonce should be unique for every 401 response for HTTP digest auth.

`opaque`
: The opaque header from the server returned unchanged by the client. It is recommended that this string be base64 or hexadecimal data. Digest auth only.

`password`
: When the authentication type is basic this is the password transmitted by the client, else *None*.

`qop`
: Indicates what "quality of protection" the client has applied to the message for HTTP digest auth.

`realm`
: This is the server realm sent back for HTTP digest auth.

`response`
: A string of 32 hex digits computed as defined in RFC 2617, which proves that the user knows a password. Digest auth only.

`uri`
: The URI from Request-URI of the Request-Line; duplicated because proxies are allowed to change the Request-Line in transit. HTTP digest auth only.

`username`
: The username transmitted. This is set for both basic and digest auth all the time.

class `werkzeug.datastructures.WWWAuthenticate`(*auth_type=None*, *values=None*, *on_update=None*)

Provides simple access to *WWW-Authenticate* headers.

`algorithm`
: A string indicating a pair of algorithms used to produce the digest and a checksum. If this is not present it is assumed to be "MD5". If the algorithm is not understood, the challenge should be ignored (and a different one used, if there is more than one).

static `auth_property`(*name*, *doc=None*)
: A static helper function for subclasses to add extra authentication system properties onto a class:

```
class FooAuthenticate(WWWAuthenticate):
    special_realm = auth_property('special_realm')
```

For more information have a look at the sourcecode to see how the regular properties (*realm* etc.) are implemented.

`domain`
: A list of URIs that define the protection space. If a URI is an absolute path, it is relative to the canonical root URL of the server being accessed.

`nonce`
: A server-specified data string which should be uniquely generated each time a 401 response is made. It is recommended that this string be base64 or hexadecimal data.

`opaque`
: A string of data, specified by the server, which should be returned by the client unchanged in the Authorization header of subsequent requests with URIs in the same protection space. It is recommended that this string be base64 or hexadecimal data.

`qop`
: A set of quality-of-privacy directives such as auth and auth-int.

`realm`
: A string to be displayed to users so they know which username and password to use. This string should contain at least the name of the host performing the authentication and might additionally indicate the collection of users who might have access.

`set_basic`(*realm='authentication required'*)
: Clear the auth info and enable basic auth.

`set_digest`(*realm*, *nonce*, *qop=('auth',)*, *opaque=None*, *algorithm=None*, *stale=False*)
Clear the auth info and enable digest auth.

`stale`
A flag, indicating that the previous request from the client was rejected because the nonce value was stale.

`to_header`()
Convert the stored values into a WWW-Authenticate header.

`type`
The type of the auth mechanism. HTTP currently specifies *Basic* and *Digest*.

class `werkzeug.datastructures.IfRange`(*etag=None*, *date=None*)
Very simple object that represents the *If-Range* header in parsed form. It will either have neither a etag or date or one of either but never both.

New in version 0.7.

`date` = **None**
The date in parsed format or *None*.

`etag` = **None**
The etag parsed and unquoted. Ranges always operate on strong etags so the weakness information is not necessary.

`to_header`()
Converts the object back into an HTTP header.

class `werkzeug.datastructures.Range`(*units*, *ranges*)
Represents a range header. All the methods are only supporting bytes as unit. It does store multiple ranges but *range_for_length()* will only work if only one range is provided.

New in version 0.7.

`make_content_range`(*length*)
Creates a *ContentRange* object from the current range and given content length.

`range_for_length`(*length*)
If the range is for bytes, the length is not None and there is exactly one range and it is satisfiable it returns a `(start, stop)` tuple, otherwise *None*.

`ranges` = **None**
A list of `(begin, end)` tuples for the range header provided. The ranges are non-inclusive.

`to_header`()
Converts the object back into an HTTP header.

`units` = **None**
The units of this range. Usually “bytes”.

class `werkzeug.datastructures.ContentRange`(*units*, *start*, *stop*, *length=None*, *on_update=None*)

Represents the content range header.

New in version 0.7.

`length`
The length of the range or *None*.

`set`(*start*, *stop*, *length=None*, *units='bytes'*)
Simple method to update the ranges.

`start`
The start point of the range or *None*.

`stop`
The stop point of the range (non-inclusive) or *None*. Can only be *None* if also start is *None*.

`units`
The units to use, usually "bytes"

`unset()`
Sets the units to *None* which indicates that the header should no longer be used.

14.3 Others

class `werkzeug.datastructures.FileStorage`(*stream=None*, *filename=None*, *name=None*, *content_type=None*, *content_length=None*, *headers=None*)

The *`FileStorage`* class is a thin wrapper over incoming files. It is used by the request object to represent uploaded files. All the attributes of the wrapper stream are proxied by the file storage so it's possible to do `storage.read()` instead of the long form `storage.stream.read()`.

`stream`
The input stream for the uploaded file. This usually points to an open temporary file.

`filename`
The filename of the file on the client.

`name`
The name of the form field.

`headers`
The multipart headers as *`Headers`* object. This usually contains irrelevant information but in combination with custom multipart requests the raw headers might be interesting.

New in version 0.6.

`close()`
: Close the underlying file if possible.

`content_length`
: The content-length sent in the header. Usually not available

`content_type`
: The content-type sent in the header. Usually not available

`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'`.

  New in version 0.7.

`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'}`.

  New in version 0.7.

`save`(*dst*, *buffer_size=16384*)
: Save the file to a destination path or file object. If the destination is a file object you have to close it yourself after the call. The buffer size is the number of bytes held in memory during the copy process. It defaults to 16KB.

  For secure file saving also have a look at `secure_filename()`.

  Parameters

  - `dst` – a filename or open file object the uploaded file is saved to.
  - `buffer_size` – the size of the buffer. This works the same as the *length* parameter of `shutil.copyfileobj()`.

CHAPTER

FIFTEEN

UTILITIES

Various utility functions shipped with Werkzeug.

15.1 HTML Helpers

class `werkzeug.utils.HTMLBuilder(`*dialect*`)`

Helper object for HTML generation.

Per default there are two instances of that class. The *html* one, and the *xhtml* one for those two dialects. The class uses keyword parameters and positional parameters to generate small snippets of HTML.

Keyword parameters are converted to XML/SGML attributes, positional arguments are used as children. Because Python accepts positional arguments before keyword arguments it's a good idea to use a list with the star-syntax for some children:

```
>>> html.p(class_='foo', *[html.a('foo', href='foo.html'), ' ',
...                        html.a('bar', href='bar.html')])
u'<p class="foo"><a href="foo.html">foo</a> <a href="bar.html">bar</a></p>'
```

This class works around some browser limitations and can not be used for arbitrary SGML/XML generation. For that purpose lxml and similar libraries exist.

Calling the builder escapes the string passed:

```
>>> html.p(html("<foo>"))
u'<p>&lt;foo&gt;</p>'
```

`werkzeug.utils.escape(`*s*, *quote=None*`)`

Replace special characters "&", "<", ">" and (") to HTML-safe sequences.

There is a special handling for *None* which escapes to an empty string.

Changed in version 0.9: *quote* is now implicitly on.

Parameters

- `s` – the string to escape.

- `quote` – ignored.

`werkzeug.utils.unescape(s)`
The reverse function of *escape*. This unescapes all the HTML entities, not only the XML entities inserted by *escape*.

Parameters `s` – the string to unescape.

15.2 General Helpers

class `werkzeug.utils.cached_property(func, name=None, doc=None)`
A decorator that converts a function into a lazy property. The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):

    @cached_property
    def foo(self):
        # calculate something important here
        return 42
```

The class has to have a *__dict__* in order for this property to work.

class `werkzeug.utils.environ_property(name, default=None, load_func=None, dump_func=None, read_only=None, doc=None)`
Maps request attributes to environment variables. This works not only for the Werzeug request object, but also any other class with an environ attribute:

```
>>> class Test(object):
...     environ = {'key': 'value'}
...     test = environ_property('key')
>>> var = Test()
>>> var.test
'value'
```

If you pass it a second value it's used as default if the key does not exist, the third one can be a converter that takes a value and converts it. If it raises `ValueError` or `TypeError` the default value is used. If no default value is provided *None* is used.

Per default the property is read only. You have to explicitly enable it by passing `read_only=False` to the constructor.

class `werkzeug.utils.header_property(name, default=None, load_func=None, dump_func=None, read_only=None, doc=None)`
Like *environ_property* but for headers.

`werkzeug.utils.parse_cookie(header, charset='utf-8', errors='replace', cls=None)`
Parse a cookie. Either from a string or WSGI environ.

Per default encoding errors are ignored. If you want a different behavior you can set *errors* to `'replace'` or `'strict'`. In strict mode a `HTTPUnicodeError` is raised.

Changed in version 0.5: This function now returns a `TypeConversionDict` instead of a regular dict. The *cls* parameter was added.

Parameters

- `header` – the header to be used to parse the cookie. Alternatively this can be a WSGI environment.
- `charset` – the charset for the cookie values.
- `errors` – the error behavior for the charset decoding.
- `cls` – an optional dict class to use. If this is not specified or *None* the default `TypeConversionDict` is used.

`werkzeug.utils.dump_cookie`(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *charset='utf-8'*, *sync_expires=True*)

Creates a new Set-Cookie header without the `Set-Cookie` prefix The parameters are the same as in the cookie Morsel object in the Python standard library but it accepts unicode data, too.

On Python 3 the return value of this function will be a unicode string, on Python 2 it will be a native string. In both cases the return value is usually restricted to ascii as the vast majority of values are properly escaped, but that is no guarantee. If a unicode string is returned it's tunneled through latin1 as required by PEP 3333.

The return value is not ASCII safe if the key contains unicode characters. This is technically against the specification but happens in the wild. It's strongly recommended to not use non-ASCII values for the keys.

Parameters

- `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. Additionally *timedelta* objects are accepted, too.
- `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` – Use this 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` – 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.

- `charset` – the encoding for unicode values.
- `sync_expires` – automatically set expires if max_age is defined but expires not.

`werkzeug.utils.redirect`(*location*, *code=302*, *Response=None*)

Returns a response object (a WSGI application) that, if called, redirects the client to the target location. Supported codes are 301, 302, 303, 305, and 307. 300 is not supported because it's not a real redirect and 304 because it's the answer for a request with a request with defined If-Modified-Since headers.

New in version 0.6: The location can now be a unicode string that is encoded using the `iri_to_uri()` function.

New in version 0.10: The class used for the Response object can now be passed in.

Parameters

- `location` – the location the response should redirect to.
- `code` – the redirect status code. defaults to 302.
- `Response` (*class*) – a Response class to use when instantiating a response. The default is *werkzeug.wrappers.Response* if unspecified.

`werkzeug.utils.append_slash_redirect`(*environ*, *code=301*)

Redirects to the same URL but with a slash appended. The behavior of this function is undefined if the path ends with a slash already.

Parameters

- `environ` – the WSGI environment for the request that triggers the redirect.
- `code` – the status code for the redirect.

`werkzeug.utils.import_string`(*import_name*, *silent=False*)

Imports an object based on a string. This is useful if you want to use import paths as endpoints or something similar. An import path can be specified either in dotted notation (`xml.sax.saxutils.escape`) or with a colon as object delimiter (`xml.sax.saxutils:escape`).

If *silent* is True the return value will be *None* if the import fails.

Parameters

- `import_name` – the dotted name for the object to import.
- `silent` – if set to *True* import errors are ignored and *None* is returned instead.

Returns imported object

werkzeug.utils.find_modules(*import_path*, *include_packages=False*, *recursive=False*)

Finds all the modules below a package. This can be useful to automatically import all views / controllers so that their metaclasses / function decorators have a chance to register themselves on the application.

Packages are not returned unless *include_packages* is *True*. This can also recursively list modules but in that case it will import all the packages to get the correct load path of that module.

Parameters

- `import_name` – the dotted name for the package to find child modules.
- `include_packages` – set to *True* if packages should be returned, too.
- `recursive` – set to *True* if recursion should happen.

Returns generator

werkzeug.utils.validate_arguments(*func*, *args*, *kwargs*, *drop_extra=True*)

Checks if the function accepts the arguments and keyword arguments. Returns a new `(args, kwargs)` tuple that can safely be passed to the function without causing a *TypeError* because the function signature is incompatible. If *drop_extra* is set to *True* (which is the default) any extra positional or keyword arguments are dropped automatically.

The exception raised provides three attributes:

missing A set of argument names that the function expected but where missing.

extra A dict of keyword arguments that the function can not handle but where provided.

extra_positional A list of values that where given by positional argument but the function cannot accept.

This can be useful for decorators that forward user submitted data to a view function:

```
from werkzeug.utils import ArgumentValidationError, validate_arguments

def sanitize(f):
    def proxy(request):
        data = request.values.to_dict()
        try:
            args, kwargs = validate_arguments(f, (request,), data)
        except ArgumentValidationError:
            raise BadRequest('The browser failed to transmit all '
                             'the data expected.')
        return f(*args, **kwargs)
    return proxy
```

Parameters

- `func` – the function the validation is performed against.
- `args` – a tuple of positional arguments.
- `kwargs` – a dict of keyword arguments.
- `drop_extra` – set to *False* if you don't want extra arguments to be silently dropped.

Returns tuple in the form `(args, kwargs)`.

`werkzeug.utils.secure_filename`(*filename*)

Pass it a filename and it will return a secure version of it. This filename can then safely be stored on a regular file system and passed to `os.path.join()`. The filename returned is an ASCII only string for maximum portability.

On windows systems the function also makes sure that the file is not named after one of the special device files.

```
>>> secure_filename("My cool movie.mov")
'My_cool_movie.mov'
>>> secure_filename("../../../etc/passwd")
'etc_passwd'
>>> secure_filename(u'i contain cool \xfcml\xe4uts.txt')
'i_contain_cool_umlauts.txt'
```

The function might return an empty filename. It's your responsibility to ensure that the filename is unique and that you generate random filename if the function returned an empty one.

New in version 0.5.

Parameters `filename` – the filename to secure

`werkzeug.utils.bind_arguments`(*func*, *args*, *kwargs*)

Bind the arguments provided into a dict. When passed a function, a tuple of arguments and a dict of keyword arguments *bind_arguments* returns a dict of names as the function would see it. This can be useful to implement a cache decorator that uses the function arguments to build the cache key based on the values of the arguments.

Parameters

- `func` – the function the arguments should be bound for.
- `args` – tuple of positional arguments.
- `kwargs` – a dict of keyword arguments.

Returns a `dict` of bound keyword arguments.

15.3 URL Helpers

Please refer to URL Helpers.

15.4 UserAgent Parsing

class `werkzeug.useragents.UserAgent(`*environ_or_string*`)`

Represents a user agent. Pass it a WSGI environment or a user agent string and you can inspect some of the details from the user agent string via the attributes. The following attributes exist:

`string`
: the raw user agent string

`platform`
: the browser platform. The following platforms are currently recognized:

  - *aix*
  - *amiga*
  - *android*
  - *bsd*
  - *chromeos*
  - *hpux*
  - *iphone*
  - *ipad*
  - *irix*
  - *linux*
  - *macos*
  - *sco*
  - *solaris*
  - *wii*
  - *windows*

`browser`
: the name of the browser. The following browsers are currently recognized:

  - *aol* *
  - *ask* *
  - *camino*
  - *chrome*

- *firefox*
- *galeon*
- *google **
- *kmeleon*
- *konqueror*
- *links*
- *lynx*
- *msie*
- *msn*
- *netscape*
- *opera*
- *safari*
- *seamonkey*
- *webkit*
- *yahoo **

(Browsers maked with a star (*) are crawlers.)

`version`
: the version of the browser

`language`
: the language of the browser

15.5 Security Helpers

New in version 0.6.1.

`werkzeug.security.generate_password_hash`(*password*, *method='pbkdf2:sha1'*, *salt_length=8*)

Hash a password with the given method and salt with with a string of the given length. The format of the string returned includes the method that was used so that *`check_password_hash()`* can check the hash.

The format for the hashed string looks like this:

```
method$salt$hash
```

This method can **not** generate unsalted passwords but it is possible to set the method to plain to enforce plaintext passwords. If a salt is used, hmac is used internally to salt the password.

If PBKDF2 is wanted it can be enabled by setting the method to `pbkdf2:method:iterations` where iterations is optional:

```
pbkdf2:sha1:2000$salt$hash
pbkdf2:sha1$salt$hash
```

Parameters

- `password` – the password to hash.
- `method` – the hash method to use (one that hashlib supports). Can optionally be in the format `pbkdf2:<method>[:iterations]` to enable PBKDF2.
- `salt_length` – the length of the salt in letters.

`werkzeug.security.check_password_hash(`*pwhash*, *password*`)`

check a password against a given salted and hashed password value. In order to support unsalted legacy passwords this method supports plain text passwords, md5 and sha1 hashes (both salted and unsalted).

Returns *True* if the password matched, *False* otherwise.

Parameters

- `pwhash` – a hashed string like returned by *`generate_password_hash()`*.
- `password` – the plaintext password to compare against the hash.

`werkzeug.security.safe_str_cmp(`*a*, *b*`)`

This function compares strings in somewhat constant time. This requires that the length of at least one string is known in advance.

Returns *True* if the two strings are equal, or *False* if they are not.

New in version 0.7.

`werkzeug.security.safe_join(`*directory*, *filename*`)`

Safely join *directory* and *filename*. If this cannot be done, this function returns `None`.

Parameters

- `directory` – the base directory.
- `filename` – the untrusted filename relative to that directory.

`werkzeug.security.pbkdf2_hex(`*data*, *salt*, *iterations=1000*, *keylen=None*, *hashfunc=None*`)`

Like *`pbkdf2_bin()`*, but returns a hex-encoded string.

New in version 0.9.

Parameters

- `data` – the data to derive.
- `salt` – the salt for the derivation.
- `iterations` – the number of iterations.
- `keylen` – the length of the resulting key. If not provided, the digest size will be used.
- `hashfunc` – the hash function to use. This can either be the string name of a known hash function, or a function from the hashlib module. Defaults to sha1.

`werkzeug.security.pbkdf2_bin`(*data*, *salt*, *iterations=1000*, *keylen=None*, *hashfunc=None*)

Returns a binary digest for the PBKDF2 hash algorithm of *data* with the given *salt*. It iterates *iterations* times and produces a key of *keylen* bytes. By default, SHA-1 is used as hash function; a different hashlib *hashfunc* can be provided.

New in version 0.9.

Parameters

- `data` – the data to derive.
- `salt` – the salt for the derivation.
- `iterations` – the number of iterations.
- `keylen` – the length of the resulting key. If not provided the digest size will be used.
- `hashfunc` – the hash function to use. This can either be the string name of a known hash function or a function from the hashlib module. Defaults to sha1.

CHAPTER

SIXTEEN

URL HELPERS

`werkzeug.urls` used to provide several wrapper functions for Python 2 urlparse, whose main purpose were to work around the behavior of the Py2 stdlib and its lack of unicode support. While this was already a somewhat inconvenient situation, it got even more complicated because Python 3's `urllib.parse` actually does handle unicode properly. In other words, this module would wrap two libraries with completely different behavior. So now this module contains a 2-and-3-compatible backport of Python 3's `urllib.parse`, which is mostly API-compatible.

class `werkzeug.urls.BaseURL`

Superclass of *`URL`* and *`BytesURL`*.

`ascii_host`

Works exactly like *`host`* but will return a result that is restricted to ASCII. If it finds a netloc that is not ASCII it will attempt to idna decode it. This is useful for socket operations when the URL might include internationalized characters.

`auth`

The authentication part in the URL if available, *None* otherwise.

`decode_netloc()`

Decodes the netloc part into a string.

`decode_query`(**args*, ***kwargs*)

Decodes the query part of the URL. Ths is a shortcut for calling *`url_decode()`* on the query argument. The arguments and keyword arguments are forwarded to *`url_decode()`* unchanged.

`get_file_location`(*pathformat=None*)

Returns a tuple with the location of the file in the form `(server, location)`. If the netloc is empty in the URL or points to localhost, it's represented as `None`.

The *pathformat* by default is autodetection but needs to be set when working with URLs of a specific system. The supported values are `'windows'` when working with Windows or DOS paths and `'posix'` when working with posix paths.

If the URL does not point to to a local file, the server and location are both represented as `None`.

Parameters `pathformat` – The expected format of the path component. Currently `'windows'` and `'posix'` are supported. Defaults to `None` which is autodetect.

`host`
: The host part of the URL if available, otherwise *None*. The host is either the hostname or the IP address mentioned in the URL. It will not contain the port.

`join(`**args*, ***kwargs*`)`
: Joins this URL with another one. This is just a convenience function for calling into *`url_join()`* and then parsing the return value again.

`password`
: The password if it was part of the URL, *None* otherwise. This undergoes URL decoding and will always be a unicode string.

`port`
: The port in the URL as an integer if it was present, *None* otherwise. This does not fill in default ports.

`raw_password`
: The password if it was part of the URL, *None* otherwise. Unlike *`password`* this one is not being decoded.

`raw_username`
: The username if it was part of the URL, *None* otherwise. Unlike *`username`* this one is not being decoded.

`replace(`***kwargs*`)`
: Return an URL with the same values, except for those parameters given new values by whichever keyword arguments are specified.

`to_iri_tuple()`
: Returns a *`URL`* tuple that holds a IRI. This will try to decode as much information as possible in the URL without losing information similar to how a web browser does it for the URL bar.

  It's usually more interesting to directly call *`uri_to_iri()`* which will return a string.

`to_uri_tuple()`
: Returns a *`BytesURL`* tuple that holds a URI. This will encode all the information in the URL properly to ASCII using the rules a web browser would follow.

  It's usually more interesting to directly call *`iri_to_uri()`* which will return a string.

`to_url()`
: Returns a URL string or bytes depending on the type of the information

stored. This is just a convenience function for calling *url_unparse()* for this URL.

`username`
: The username if it was part of the URL, *None* otherwise. This undergoes URL decoding and will always be a unicode string.

class `werkzeug.urls.BytesURL`
: Represents a parsed URL in bytes.

`decode`(*charset='utf-8'*, *errors='replace'*)
: Decodes the URL to a tuple made out of strings. The charset is only being used for the path, query and fragment.

`encode_netloc()`
: Returns the netloc unchanged as bytes.

class `werkzeug.urls.Href`(*base='./'*, *charset='utf-8'*, *sort=False*, *key=None*)
: Implements a callable that constructs URLs with the given base. The function can be called with any number of positional and keyword arguments which than are used to assemble the URL. Works with URLs and posix paths.

  Positional arguments are appended as individual segments to the path of the URL:

```
>>> href = Href('/foo')
>>> href('bar', 23)
'/foo/bar/23'
>>> href('foo', bar=23)
'/foo/foo?bar=23'
```

  If any of the arguments (positional or keyword) evaluates to *None* it will be skipped. If no keyword arguments are given the last argument can be a `dict` or `MultiDict` (or any other dict subclass), otherwise the keyword arguments are used for the query parameters, cutting off the first trailing underscore of the parameter name:

```
>>> href(is_=42)
'/foo?is=42'
>>> href({'foo': 'bar'})
'/foo?foo=bar'
```

  Combining of both methods is not allowed:

```
>>> href({'foo': 'bar'}, bar=42)
Traceback (most recent call last):
  ...
TypeError: keyword arguments and query-dicts can't be combined
```

  Accessing attributes on the href object creates a new href object with the attribute name as prefix:

```
>>> bar_href = href.bar
>>> bar_href("blub")
'/foo/bar/blub'
```

If *sort* is set to *True* the items are sorted by *key* or the default sorting algorithm:

```
>>> href = Href("/", sort=True)
>>> href(a=1, b=2, c=3)
'/?a=1&b=2&c=3'
```

New in version 0.5: *sort* and *key* were added.

class `werkzeug.urls.URL`

Represents a parsed URL. This behaves like a regular tuple but also has some extra attributes that give further insight into the URL.

`encode`(*charset='utf-8'*, *errors='replace'*)

Encodes the URL to a tuple made out of bytes. The charset is only being used for the path, query and fragment.

`encode_netloc()`

Encodes the netloc part to an ASCII safe URL as bytes.

`werkzeug.urls.iri_to_uri`(*iri*, *charset='utf-8'*, *errors='strict'*, *safe_conversion=False*)

Converts any unicode based IRI to an acceptable ASCII URI. Werkzeug always uses utf-8 URLs internally because this is what browsers and HTTP do as well. In some places where it accepts an URL it also accepts a unicode IRI and converts it into a URI.

Examples for IRI versus URI:

```
>>> iri_to_uri(u'http://                N{SNOWMAN}.net/')
'http://xn--n3h.net/'
>>> iri_to_uri(u'http://üser:pässword@        N{SNOWMAN}.net/påth')
'http://%C3%BCser:p%C3%A4ssword@xn--n3h.net/p%C3%A5th'
```

There is a general problem with IRI and URI conversion with some protocols that appear in the wild that are in violation of the URI specification. In places where Werkzeug goes through a forced IRI to URI conversion it will set the *safe_conversion* flag which will not perform a conversion if the end result is already ASCII. This can mean that the return value is not an entirely correct URI but it will not destroy such invalid URLs in the process.

As an example consider the following two IRIs:

```
magnet:?xt=uri:whatever
itms-services://?action=download-manifest
```

The internal representation after parsing of those URLs is the same and there is no way to reconstruct the original one. If safe conversion is enabled however this function becomes a noop for both of those strings as they both can be considered URIs.

New in version 0.6.

Changed in version 0.9.6: The *safe_conversion* parameter was added.

Parameters

- `iri` – The IRI to convert.
- `charset` – The charset for the URI.
- `safe_conversion` – indicates if a safe conversion should take place. For more information see the explanation above.

`werkzeug.urls.uri_to_iri`(*uri*, *charset='utf-8'*, *errors='replace'*)

Converts a URI in a given charset to a IRI.

Examples for URI versus IRI:

```
>>> uri_to_iri(b'http://xn--n3h.net/')
u'http://\u2603.net/'
>>> uri_to_iri(b'http://%C3%BCser:p%C3%A4ssword@xn--n3h.net/p%C3%A5th')
u'http://\xfcser:p\xe4ssword@\u2603.net/p\xe5th'
```

Query strings are left unchanged:

```
>>> uri_to_iri('/?foo=24&x=%26%2f')
u'/?foo=24&x=%26%2f'
```

New in version 0.6.

Parameters

- `uri` – The URI to convert.
- `charset` – The charset of the URI.
- `errors` – The error handling on decode.

`werkzeug.urls.url_decode`(*s*, *charset='utf-8'*, *decode_keys=False*, *include_empty=True*, *errors='replace'*, *separator='&'*, *cls=None*)

Parse a querystring and return it as `MultiDict`. There is a difference in key decoding on different Python versions. On Python 3 keys will always be fully decoded whereas on Python 2, keys will remain bytestrings if they fit into ASCII. On 2.x keys can be forced to be unicode by setting *decode_keys* to *True*.

If the charset is set to *None* no unicode decoding will happen and raw bytes will be returned.

Per default a missing value for a key will default to an empty key. If you don't want that behavior you can set *include_empty* to *False*.

Per default encoding errors are ignored. If you want a different behavior you can set *errors* to `'replace'` or `'strict'`. In strict mode a *HTTPUnicodeError* is raised.

Changed in version 0.5: In previous versions ";" and "&" could be used for url decoding. This changed in 0.5 where only "&" is supported. If you want to use ";" instead a different *separator* can be provided.

The *cls* parameter was added.

Parameters

- `s` – a string with the query string to decode.
- `charset` – the charset of the query string. If set to *None* no unicode decoding will take place.
- `decode_keys` – Used on Python 2.x to control whether keys should be forced to be unicode objects. If set to *True* then keys will be unicode in all cases. Otherwise, they remain *str* if they fit into ASCII.
- `include_empty` – Set to *False* if you don't want empty values to appear in the dict.
- `errors` – the decoding error behavior.
- `separator` – the pair separator to be used, defaults to `&`
- `cls` – an optional dict class to use. If this is not specified or *None* the default `MultiDict` is used.

`werkzeug.urls.url_decode_stream`(*stream*, *charset='utf-8'*, *decode_keys=False*, *include_empty=True*, *errors='replace'*, *separator='&'*, *cls=None*, *limit=None*, *return_iterator=False*)

Works like *`url_decode()`* but decodes a stream. The behavior of stream and limit follows functions like *`make_line_iter()`*. The generator of pairs is directly fed to the *cls* so you can consume the data while it's parsed.

New in version 0.8.

Parameters

- `stream` – a stream with the encoded querystring
- `charset` – the charset of the query string. If set to *None* no unicode decoding will take place.
- `decode_keys` – Used on Python 2.x to control whether keys should be forced to be unicode objects. If set to *True*, keys will be unicode in all cases. Otherwise, they remain *str* if they fit into ASCII.
- `include_empty` – Set to *False* if you don't want empty values to appear in the dict.
- `errors` – the decoding error behavior.
- `separator` – the pair separator to be used, defaults to `&`
- `cls` – an optional dict class to use. If this is not specified or *None* the default `MultiDict` is used.
- `limit` – the content length of the URL data. Not necessary if a limited stream is provided.

- `return_iterator` – if set to *True* the *cls* argument is ignored and an iterator over all decoded pairs is returned

`werkzeug.urls.url_encode`(*obj*, *charset='utf-8'*, *encode_keys=False*, *sort=False*, *key=None*, *separator=b'&'*)

URL encode a dict/*MultiDict*. If a value is *None* it will not appear in the result string. Per default only values are encoded into the target charset strings. If *encode_keys* is set to `True` unicode keys are supported too.

If *sort* is set to *True* the items are sorted by *key* or the default sorting algorithm.

New in version 0.5: *sort*, *key*, and *separator* were added.

Parameters

- `obj` – the object to encode into a query string.
- `charset` – the charset of the query string.
- `encode_keys` – set to *True* if you have unicode keys. (Ignored on Python 3.x)
- `sort` – set to *True* if you want parameters to be sorted by *key*.
- `separator` – the separator to be used for the pairs.
- `key` – an optional function to be used for sorting. For more details check out the `sorted()` documentation.

`werkzeug.urls.url_encode_stream`(*obj*, *stream=None*, *charset='utf-8'*, *encode_keys=False*, *sort=False*, *key=None*, *separator=b'&'*)

Like *`url_encode()`* but writes the results to a stream object. If the stream is *None* a generator over all encoded pairs is returned.

New in version 0.8.

Parameters

- `obj` – the object to encode into a query string.
- `stream` – a stream to write the encoded object into or *None* if an iterator over the encoded pairs should be returned. In that case the separator argument is ignored.
- `charset` – the charset of the query string.
- `encode_keys` – set to *True* if you have unicode keys. (Ignored on Python 3.x)
- `sort` – set to *True* if you want parameters to be sorted by *key*.
- `separator` – the separator to be used for the pairs.
- `key` – an optional function to be used for sorting. For more details check out the `sorted()` documentation.

`werkzeug.urls.url_fix`(*s*, *charset='utf-8'*)

Sometimes you get an URL by a user that just isn't a real URL because it contains

unsafe characters like ‘ ‘ and so on. This function can fix some of the problems in a similar way browsers handle data entered by the user:

```
>>> url_fix(u'http://de.wikipedia.org/wiki/Elf (Begriffskl\xe4rung)')
'http://de.wikipedia.org/wiki/Elf%20(Begriffskl%C3%A4rung)'
```

Parameters

- `s` – the string with the URL to fix.
- `charset` – The target charset for the URL if the url was given as unicode string.

`werkzeug.urls.url_join`(*base*, *url*, *allow_fragments=True*)

Join a base URL and a possibly relative URL to form an absolute interpretation of the latter.

Parameters

- `base` – the base URL for the join operation.
- `url` – the URL to join.
- `allow_fragments` – indicates whether fragments should be allowed.

`werkzeug.urls.url_parse`(*url*, *scheme=None*, *allow_fragments=True*)

Parses a URL from a string into a *URL* tuple. If the URL is lacking a scheme it can be provided as second argument. Otherwise, it is ignored. Optionally fragments can be stripped from the URL by setting *allow_fragments* to *False*.

The inverse of this function is *url_unparse()*.

Parameters

- `url` – the URL to parse.
- `scheme` – the default schema to use if the URL is schemaless.
- `allow_fragments` – if set to *False* a fragment will be removed from the URL.

`werkzeug.urls.url_quote`(*string*, *charset='utf-8'*, *errors='strict'*, *safe='/:'*, *unsafe=''*)

URL encode a single string with a given encoding.

Parameters

- `s` – the string to quote.
- `charset` – the charset to be used.
- `safe` – an optional sequence of safe characters.
- `unsafe` – an optional sequence of unsafe characters.

New in version 0.9.2: The *unsafe* parameter was added.

werkzeug.urls.url_quote_plus(*string*, *charset='utf-8'*, *errors='strict'*, *safe=''*)

URL encode a single string with the given encoding and convert whitespace to "+".

Parameters

- s – The string to quote.
- charset – The charset to be used.
- safe – An optional sequence of safe characters.

werkzeug.urls.url_unparse(*components*)

The reverse operation to *url_parse()*. This accepts arbitrary as well as *URL* tuples and returns a URL as a string.

Parameters components – the parsed URL as tuple which should be converted into a URL string.

werkzeug.urls.url_unquote(*string*, *charset='utf-8'*, *errors='replace'*, *unsafe=''*)

URL decode a single string with a given encoding. If the charset is set to *None* no unicode decoding is performed and raw bytes are returned.

Parameters

- s – the string to unquote.
- charset – the charset of the query string. If set to *None* no unicode decoding will take place.
- errors – the error handling for the charset decoding.

werkzeug.urls.url_unquote_plus(*s*, *charset='utf-8'*, *errors='replace'*)

URL decode a single string with the given *charset* and decode "+" to whitespace.

Per default encoding errors are ignored. If you want a different behavior you can set *errors* to 'replace' or 'strict'. In strict mode a HTTPUnicodeError is raised.

Parameters

- s – The string to unquote.
- charset – the charset of the query string. If set to *None* no unicode decoding will take place.
- errors – The error handling for the *charset* decoding.

CHAPTER

SEVENTEEN

CONTEXT LOCALS

Sooner or later you have some things you want to have in every single view or helper function or whatever. In PHP the way to go are global variables. However, that isn't possible in WSGI applications without a major drawback: As soon as you operate on the global namespace your application isn't thread-safe any longer.

The Python standard library comes with a utility called "thread locals". A thread local is a global object in which you can put stuff in and get back later in a thread-safe way. That means whenever you set or get an object on a thread local object, the thread local object checks in which thread you are and retrieves the correct value.

This, however, has a few disadvantages. For example, besides threads there are other ways to handle concurrency in Python. A very popular approach is greenlets. Also, whether every request gets its own thread is not guaranteed in WSGI. It could be that a request is reusing a thread from before, and hence data is left in the thread local object.

Here's a simple example of how one could use werkzeug.local:

```
from werkzeug.local import Local, LocalManager

local = Local()
local_manager = LocalManager([local])

def application(environ, start_response):
    local.request = request = Request(environ)
    ...

application = local_manager.make_middleware(application)
```

This binds the request to *local.request*. Every other piece of code executed after this assignment in the same context can safely access local.request and will get the same request object. The *make_middleware* method on the local manager ensures that all references to the local objects are cleared up after the request.

The same context means the same greenlet (if you're using greenlets) in the same thread and same process.

If a request object is not yet set on the local object and you try to access it, you will get an *AttributeError*. You can use *getattr* to avoid that:

```
def get_request():
    return getattr(local, 'request', None)
```

This will try to get the request or return *None* if the request is not (yet?) available.

Note that local objects cannot manage themselves, for that you need a local manager. You can pass a local manager multiple locals or add additionals later by appending them to *manager.locals* and everytime the manager cleans up it will clean up all the data left in the locals for this context.

`werkzeug.local.release_local(`*local*`)`

Releases the contents of the local for the current context. This makes it possible to use locals without a manager.

Example:

```
>>> loc = Local()
>>> loc.foo = 42
>>> release_local(loc)
>>> hasattr(loc, 'foo')
False
```

With this function one can release `Local` objects as well as *`LocalStack`* objects. However it is not possible to release data held by proxies that way, one always has to retain a reference to the underlying local object in order to be able to release it.

New in version 0.6.1.

class `werkzeug.local.LocalManager(`*locals=None*, *ident_func=None*`)`

Local objects cannot manage themselves. For that you need a local manager. You can pass a local manager multiple locals or add them later by appending them to *manager.locals*. Everytime the manager cleans up it, will clean up all the data left in the locals for this context.

The *ident_func* parameter can be added to override the default ident function for the wrapped locals.

Changed in version 0.6.1: Instead of a manager the *`release_local()`* function can be used as well.

Changed in version 0.7: *ident_func* was added.

`cleanup()`

Manually clean up the data in the locals for this context. Call this at the end of the request or use *make_middleware()*.

`get_ident()`

Return the context identifier the local objects use internally for this context. You cannot override this method to change the behavior but use it to link other context local objects (such as SQLAlchemy's scoped sessions) to the Werkzeug locals.

Changed in version 0.7: You can pass a different ident function to the local

manager that will then be propagated to all the locals passed to the constructor.

`make_middleware(`*app*`)`
Wrap a WSGI application so that cleaning up happens after request end.

`middleware(`*func*`)`
Like *make_middleware* but for decorating functions.

Example usage:

```
@manager.middleware
def application(environ, start_response):
    ...
```

The difference to *make_middleware* is that the function passed will have all the arguments copied from the inner application (name, docstring, module).

class `werkzeug.local.LocalStack`
This class works similar to a `Local` but keeps a stack of objects instead. This is best explained with an example:

```
>>> ls = LocalStack()
>>> ls.push(42)
>>> ls.top
42
>>> ls.push(23)
>>> ls.top
23
>>> ls.pop()
23
>>> ls.top
42
```

They can be force released by using a *LocalManager* or with the *release_local()* function but the correct way is to pop the item from the stack after using. When the stack is empty it will no longer be bound to the current context (and as such released).

By calling the stack without arguments it returns a proxy that resolves to the topmost item on the stack.

New in version 0.6.1.

`pop()`
Removes the topmost item from the stack, will return the old value or *None* if the stack was already empty.

`push(`*obj*`)`
Pushes a new item to the stack

`top`
The topmost item on the stack. If the stack is empty, *None* is returned.

class werkzeug.local.LocalProxy(*local*, *name=None*)

Acts as a proxy for a werkzeug local. Forwards all operations to a proxied object. The only operations not supported for forwarding are right handed operands and any kind of assignment.

Example usage:

```
from werkzeug.local import Local
l = Local()

# these are proxies
request = l('request')
user = l('user')


from werkzeug.local import LocalStack
_response_local = LocalStack()

# this is a proxy
response = _response_local()
```

Whenever something is bound to l.user / l.request the proxy objects will forward all operations. If no object is bound a RuntimeError will be raised.

To create proxies to Local or *LocalStack* objects, call the object as shown above. If you want to have a proxy to an object looked up by a function, you can (as of Werkzeug 0.6.1) pass a function to the *LocalProxy* constructor:

```
session = LocalProxy(lambda: get_current_request().session)
```

Changed in version 0.6.1: The class can be instanciated with a callable as well now.

Keep in mind that repr() is also forwarded, so if you want to find out if you are dealing with a proxy you can do an isinstance() check:

```
>>> from werkzeug.local import LocalProxy
>>> isinstance(request, LocalProxy)
True
```

You can also create proxy objects by hand:

```
from werkzeug.local import Local, LocalProxy
local = Local()
request = LocalProxy(local, 'request')
```

_get_current_object()

Return the current object. This is useful if you want the real object behind the proxy at a time for performance reasons or because you want to pass the object into a different context.

CHAPTER

EIGHTEEN

MIDDLEWARES

Middlewares wrap applications to dispatch between then or provide additional request handling. Additionally to the middlewares documented here, there is also the `DebuggedApplication` class that is implemented as a WSGI middleware.

class `werkzeug.wsgi.SharedDataMiddleware`(*app*, *exports*, *disallow=None*, *cache=True*, *cache_timeout=43200*, *fallback_mimetype='text/plain'*)

A WSGI middleware that provides static content for development environments or simple server setups. Usage is quite simple:

```
import os
from werkzeug.wsgi import SharedDataMiddleware

app = SharedDataMiddleware(app, {
    '/shared': os.path.join(os.path.dirname(__file__), 'shared')
})
```

The contents of the folder `./shared` will now be available on `http://example.com/shared/`. This is pretty useful during development because a standalone media server is not required. One can also mount files on the root folder and still continue to use the application because the shared data middleware forwards all unhandled requests to the application, even if the requests are below one of the shared folders.

If *pkg_resources* is available you can also tell the middleware to serve files from package data:

```
app = SharedDataMiddleware(app, {
    '/shared': ('myapplication', 'shared_files')
})
```

This will then serve the `shared_files` folder in the *myapplication* Python package.

The optional *disallow* parameter can be a list of `fnmatch()` rules for files that are not accessible from the web. If *cache* is set to *False* no caching headers are sent.

Currently the middleware does not support non ASCII filenames. If the encoding on the file system happens to be the encoding of the URI it may work but this

could also be by accident. We strongly suggest using ASCII only file names for static files.

The middleware will guess the mimetype using the Python *mimetype* module. If it's unable to figure out the charset it will fall back to *fallback_mimetype*.

Changed in version 0.5: The cache timeout is configurable now.

New in version 0.6: The *fallback_mimetype* parameter was added.

Parameters

- `app` – the application to wrap. If you don't want to wrap an application you can pass it `NotFound`.
- `exports` – a dict of exported files and folders.
- `disallow` – a list of `fnmatch()` rules.
- `fallback_mimetype` – the fallback mimetype for unknown files.
- `cache` – enable or disable caching headers.
- `cache_timeout` – the cache timeout in seconds for the headers.

`is_allowed`(*filename*)
Subclasses can override this method to disallow the access to certain files. However by providing *disallow* in the constructor this method is overwritten.

class `werkzeug.wsgi.DispatcherMiddleware`(*app*, *mounts=None*)
Allows one to mount middlewares or applications in a WSGI application. This is useful if you want to combine multiple WSGI applications:

```
app = DispatcherMiddleware(app, {
    '/app2':        app2,
    '/app3':        app3
})
```

Also there's the ...

`werkzeug._internal._easteregg`(*app=None*)
Like the name says. But who knows how it works?

CHAPTER

NINETEEN

HTTP EXCEPTIONS

This module implements a number of Python exceptions you can raise from within your views to trigger a standard non-200 response.

19.1 Usage Example

```
from werkzeug.wrappers import BaseRequest
from werkzeug.wsgi import responder
from werkzeug.exceptions import HTTPException, NotFound

def view(request):
    raise NotFound()

@responder
def application(environ, start_response):
    request = BaseRequest(environ)
    try:
        return view(request)
    except HTTPException as e:
        return e
```

As you can see from this example those exceptions are callable WSGI applications. Because of Python 2.4 compatibility those do not extend from the response objects but only from the python exception class.

As a matter of fact they are not Werkzeug response objects. However you can get a response object by calling `get_response()` on a HTTP exception.

Keep in mind that you have to pass an environment to `get_response()` because some errors fetch additional information from the WSGI environment.

If you want to hook in a different exception page to say, a 404 status code, you can add a second except for a specific subclass of an error:

```
@responder
def application(environ, start_response):
    request = BaseRequest(environ)
    try:
```

```
        return view(request)
    except NotFound, e:
        return not_found(request)
    except HTTPException, e:
        return e
```

19.2 Error Classes

The following error classes exist in Werkzeug:

exception `werkzeug.exceptions.BadRequest`(*description=None*, *response=None*)
400 Bad Request

Raise if the browser sends something to the application the application or server cannot handle.

exception `werkzeug.exceptions.Unauthorized`(*description=None*, *response=None*)
401 Unauthorized

Raise if the user is not authorized. Also used if you want to use HTTP basic auth.

exception `werkzeug.exceptions.Forbidden`(*description=None*, *response=None*)
403 Forbidden

Raise if the user doesn't have the permission for the requested resource but was authenticated.

exception `werkzeug.exceptions.NotFound`(*description=None*, *response=None*)
404 Not Found

Raise if a resource does not exist and never existed.

exception `werkzeug.exceptions.MethodNotAllowed`(*valid_methods=None*, *description=None*)
405 Method Not Allowed

Raise if the server used a method the resource does not handle. For example *POST* if the resource is view only. Especially useful for REST.

The first argument for this exception should be a list of allowed methods. Strictly speaking the response would be invalid if you don't provide valid methods in the header which you can do with that list.

exception `werkzeug.exceptions.NotAcceptable`(*description=None*, *response=None*)
406 Not Acceptable

Raise if the server can't return any content conforming to the *Accept* headers of the client.

exception `werkzeug.exceptions.RequestTimeout`(*description=None*, *response=None*)

408 Request Timeout

Raise to signalize a timeout.

exception `werkzeug.exceptions.Conflict`(*description=None*, *response=None*)

409 Conflict

Raise to signal that a request cannot be completed because it conflicts with the current state on the server.

New in version 0.7.

exception `werkzeug.exceptions.Gone`(*description=None*, *response=None*)

410 Gone

Raise if a resource existed previously and went away without new location.

exception `werkzeug.exceptions.LengthRequired`(*description=None*, *response=None*)

411 Length Required

Raise if the browser submitted data but no `Content-Length` header which is required for the kind of processing the server does.

exception `werkzeug.exceptions.PreconditionFailed`(*description=None*, *response=None*)

412 Precondition Failed

Status code used in combination with `If-Match`, `If-None-Match`, or `If-Unmodified-Since`.

exception `werkzeug.exceptions.RequestEntityTooLarge`(*description=None*, *response=None*)

413 Request Entity Too Large

The status code one should return if the data submitted exceeded a given limit.

exception `werkzeug.exceptions.RequestURITooLarge`(*description=None*, *response=None*)

414 Request URI Too Large

Like *413* but for too long URLs.

exception `werkzeug.exceptions.UnsupportedMediaType`(*description=None*, *response=None*)

415 Unsupported Media Type

The status code returned if the server is unable to handle the media type the client transmitted.

exception `werkzeug.exceptions.RequestedRangeNotSatisfiable`(*description=None*, *response=None*)

416 Requested Range Not Satisfiable

The client asked for a part of the file that lies beyond the end of the file.

New in version 0.7.

exception `werkzeug.exceptions.ExpectationFailed`(*description=None*, *response=None*)

417 Expectation Failed

The server cannot meet the requirements of the Expect request-header.

New in version 0.7.

exception `werkzeug.exceptions.ImATeapot`(*description=None*, *response=None*)

418 I'm a teapot

The server should return this if it is a teapot and someone attempted to brew coffee with it.

New in version 0.7.

exception `werkzeug.exceptions.PreconditionRequired`(*description=None*, *response=None*)

428 Precondition Required

The server requires this request to be conditional, typically to prevent the lost update problem, which is a race condition between two or more clients attempting to update a resource through PUT or DELETE. By requiring each client to include a conditional header ("If-Match" or "If-Unmodified- Since") with the proper value retained from a recent GET request, the server ensures that each client has at least seen the previous revision of the resource.

exception `werkzeug.exceptions.TooManyRequests`(*description=None*, *response=None*)

429 Too Many Requests

The server is limiting the rate at which this user receives responses, and this request exceeds that rate. (The server may use any convenient method to identify users and their request rates). The server may include a "Retry-After" header to indicate how long the user should wait before retrying.

exception `werkzeug.exceptions.RequestHeaderFieldsTooLarge`(*description=None*, *response=None*)

431 Request Header Fields Too Large

The server refuses to process the request because the header fields are too large. One or more individual fields may be too large, or the set of all headers is too large.

exception `werkzeug.exceptions.InternalServerError`(*description=None*, *response=None*)

500 Internal Server Error

Raise if an internal server error occurred. This is a good fallback if an unknown error occurred in the dispatcher.

exception `werkzeug.exceptions.NotImplemented`(*description=None*, *response=None*)

501 Not Implemented

Raise if the application does not support the action requested by the browser.

exception `werkzeug.exceptions.BadGateway`(*description=None*, *response=None*)

502 Bad Gateway

If you do proxying in your application you should return this status code if you received an invalid response from the upstream server it accessed in attempting to fulfill the request.

exception `werkzeug.exceptions.ServiceUnavailable`(*description=None*, *response=None*)

503 Service Unavailable

Status code you should return if a service is temporarily unavailable.

exception `werkzeug.exceptions.HTTPUnicodeError`

This exception is used to signal unicode decode errors of request data. For more information see the *Unicode* chapter.

exception `werkzeug.exceptions.ClientDisconnected`(*description=None*, *response=None*)

Internal exception that is raised if Werkzeug detects a disconnected client. Since the client is already gone at that point attempting to send the error message to the client might not work and might ultimately result in another exception in the server. Mainly this is here so that it is silenced by default as far as Werkzeug is concerned.

Since disconnections cannot be reliably detected and are unspecified by WSGI to a large extent this might or might not be raised if a client is gone.

New in version 0.8.

exception `werkzeug.exceptions.SecurityError`(*description=None*, *response=None*)

Raised if something triggers a security error. This is otherwise exactly like a bad request error.

New in version 0.9.

19.3 Baseclass

All the exceptions implement this common interface:

exception `werkzeug.exceptions.HTTPException`(*description=None*, *response=None*)

Baseclass for all HTTP exceptions. This exception can be called as WSGI application to render a default error page or you can catch the subclasses of it independently and render nicer error messages.

`__call__`(*environ*, *start_response*)
Call the exception as WSGI application.

Parameters

- `environ` – the WSGI environment.
- `start_response` – the response callable provided by the WSGI server.

`get_response`(*environ=None*)
Get a response object. If one was passed to the exception it's returned directly.

Parameters `environ` – the optional environ for the request. This can be used to modify the response depending on how the request looked like.

Returns a `Response` object or a subclass thereof.

19.4 Special HTTP Exceptions

Starting with Werkzeug 0.3 some of the builtin classes raise exceptions that look like regular python exceptions (eg `KeyError`) but are *`BadRequest`* HTTP exceptions at the same time. This decision was made to simplify a common pattern where you want to abort if the client tampered with the submitted form data in a way that the application can't recover properly and should abort with `400 BAD REQUEST`.

Assuming the application catches all HTTP exceptions and reacts to them properly a view function could do the following savely and doesn't have to check if the keys exist:

```
def new_post(request):
    post = Post(title=request.form['title'], body=request.form['body'])
    post.save()
    return redirect(post.url)
```

If *title* or *body* are missing in the form a special key error will be raised which behaves like a `KeyError` but also a *`BadRequest`* exception.

19.5 Simple Aborting

Sometimes it's convenient to just raise an exception by the error code, without importing the exception and looking up the name etc. For this purpose there is the *`abort()`* function.

`werkzeug.exceptions.abort`(*status*)
It 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 from above 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'))
```

If you want to use this functionality with custom exceptions you can create an instance of the aborter class:

class `werkzeug.exceptions.Aborter`(*mapping=None*, *extra=None*)
When passed a dict of code -> exception items it can be used as callable that raises exceptions. If the first argument to the callable is an integer it will be looked up in the mapping, if it's a WSGI application it will be raised in a proxy exception.

The rest of the arguments are forwarded to the exception constructor.

19.6 Custom Errors

As you can see from the list above not all status codes are available as errors. Especially redirects and ather non 200 status codes that represent do not represent errors are missing. For redirects you can use the `redirect()` function from the utilities.

If you want to add an error yourself you can subclass *`HTTPException`*:

```
from werkzeug.exceptions import HTTPException

class PaymentRequired(HTTPException):
    code = 402
    description = '<p>Payment required.</p>'
```

This is the minimal code you need for your own exception. If you want to add more logic to the errors you can override the `get_description()`, `get_body()`, `get_headers()` and *`get_response()`* methods. In any case you should have a look at the sourcecode of the exceptions module.

You can override the default description in the constructor with the *description* parameter (it's the first argument for all exceptions except of the *`MethodNotAllowed`* which accepts a list of allowed methods as first argument):

```
raise BadRequest('Request failed because X was not present')
```

Part IV
DEPLOYMENT

This section covers running your application in production on a web server such as Apache or lighttpd.

CHAPTER

TWENTY

APPLICATION DEPLOYMENT

This section covers running your application in production on a web server such as Apache or lighttpd.

20.1 CGI

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

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

20.1.1 Creating a *.cgi* file

First you need to create the CGI application file. Let's call it *yourapplication.cgi*:

```
#!/usr/bin/python
from wsgiref.handlers import CGIHandler
from yourapplication import make_app

application = make_app()
CGIHandler().run(application)
```

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

20.1.2 Server Setup

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

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

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

For more information consult the documentation of your webserver.

20.2 *mod_wsgi* (Apache)

If you are using the Apache webserver you should consider using mod_wsgi.

20.2.1 Installing *mod_wsgi*

If you don't have *mod_wsgi* installed yet you have to either install it using a package manager or compile it yourself.

The mod_wsgi installation instructions cover installation instructions for source installations on UNIX systems.

If you are using ubuntu / debian you can apt-get it and activate it as follows:

```
# apt-get install libapache2-mod-wsgi
```

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

```
# pkg_add -r mod_wsgi
```

If you are using pkgsrc you can install *mod_wsgi* by compiling the *www/ap2-wsgi* package.

If you encounter segfaulting child processes after the first apache reload you can safely ignore them. Just restart the server.

20.2.2 Creating a *.wsgi* file

To run your application you need a *yourapplication.wsgi* file. This file contains the code *mod_wsgi* is executing on startup to get the application object. The object called *application* in that file is then used as application.

For most applications the following file should be sufficient:

```
from yourapplication import make_app
application = make_app()
```

If you don't have a factory function for application creation but a singleton instance you can directly import that one as *application*.

Store that file somewhere where you will find it again (eg: */var/www/yourapplication*) and make sure that *yourapplication* and all the libraries that are in use are on the python load path. If you don't want to install it system wide consider using a virtual python instance.

20.2.3 Configuring Apache

The last thing you have to do is to create an Apache configuration file for your application. In this example we are telling *mod_wsgi* to execute the application under a different user for security reasons:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess yourapplication user=user1 group=group1 processes=2 threads=5
    WSGIScriptAlias / /var/www/yourapplication/yourapplication.wsgi

    <Directory /var/www/yourapplication>
        WSGIProcessGroup yourapplication
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

For more information consult the mod_wsgi wiki.

20.3 FastCGI

A very popular deployment setup on servers like lighttpd and nginx is FastCGI. To use your WSGI application with any of them you will need a FastCGI server first.

The most popular one is flup which we will use for this guide. Make sure to have it installed.

20.3.1 Creating a *.fcgi* file

First you need to create the FastCGI server file. Let's call it *yourapplication.fcgi*:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from yourapplication import make_app

if __name__ == '__main__':
    application = make_app()
    WSGIServer(application).run()
```

This is enough for Apache to work, however ngingx 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
```

20.3.2 Configuring lighttpd

A basic FastCGI configuration for lighttpd looks like this:

```
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).

20.3.3 Configuring nginx

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

A basic FastCGI configuration for nginx looks like this:

```
location /yourapplication/ {
    include fastcgi_params;
    if ($uri ~ ^/yourapplication/(.*)?) {
        set $path_url $1;
    }
```

```
    fastcgi_param PATH_INFO $path_url;
    fastcgi_param SCRIPT_NAME /yourapplication;
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

This configuration binds the application to */yourapplication*. If you want to have it in the URL root it's a bit easier because you don't have to figure out how to calculate *PATH_INFO* and *SCRIPT_NAME*:

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

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

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

20.3.4 Debugging

FastCGI deployments tend to be hard to debug on most webservers. Very often the only thing the server log tells you is something along the lines of "premature end of headers". In order to debug the application the only thing that can really give you ideas why it breaks is switching to the correct user and executing the application by hand.

This example assumes your application is called *application.fcgi* and that your webserver user is *www-data*:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcg", line 4, in <module>
ImportError: No module named yourapplication
```

In this case the error seems to be "yourapplication" not being on the python path. Common problems are:

- relative paths being used. Don't rely on the current working directory
- the code depending on environment variables that are not set by the web server.
- different python interpreters being used.

20.4 HTTP Proxying

Many people prefer using a standalone Python HTTP server and proxying that server via nginx, Apache etc.

A very stable Python server is CherryPy. This part of the documentation shows you how to combine your WSGI application with the CherryPy WSGI server and how to configure the webserver for proxying.

20.4.1 Creating a *.py* server

To run your application you need a *start-server.py* file that starts up the WSGI Server.

It looks something along these lines:

```
from cherrypy import wsgiserver
from yourapplication import make_app
server = wsgiserver.CherryPyWSGIServer(('localhost', 8080), make_app())
try:
    server.start()
except KeyboardInterrupt:
    server.stop()
```

If you now start the file the server will listen on *localhost:8080*. Keep in mind that WSGI applications behave slightly different for proxied setups. If you have not developed your application for proxying in mind, you can apply the *ProxyFix* middleware.

20.4.2 Configuring nginx

As an example we show here how to configure nginx to proxy to the server.

The basic nginx configuration looks like this:

```
location / {
    proxy_set_header        Host $host;
    proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_pass              http://127.0.0.1:8080;
    proxy_redirect          default;
}
```

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

```
$ screen
$ python start-server.py
```

Part V
CONTRIBUTED MODULES

A lot of useful code contributed by the community is shipped with Werkzeug as part of the *contrib* module:

CHAPTER

TWENTYONE

CONTRIBUTED MODULES

A lot of useful code contributed by the community is shipped with Werkzeug as part of the *contrib* module:

21.1 Atom Syndication

This module provides a class called *AtomFeed* which can be used to generate feeds in the Atom syndication format (see **RFC 4287**).

Example:

```
def atom_feed(request):
    feed = AtomFeed("My Blog", feed_url=request.url,
                    url=request.host_url,
                    subtitle="My example blog for a feed test.")
    for post in Post.query.limit(10).all():
        feed.add(post.title, post.body, content_type='html',
                 author=post.author, url=post.url, id=post.uid,
                 updated=post.last_update, published=post.pub_date)
    return feed.get_response()
```

class `werkzeug.contrib.atom.AtomFeed`(*title=None*, *entries=None*, ***kwargs*)

A helper class that creates Atom feeds.

Parameters

- `title` – the title of the feed. Required.
- `title_type` – the type attribute for the title element. One of `'html'`, `'text'` or `'xhtml'`.
- `url` – the url for the feed (not the url *of* the feed)
- `id` – a globally unique id for the feed. Must be an URI. If not present the *feed_url* is used, but one of both is required.
- `updated` – the time the feed was modified the last time. Must be a `datetime.datetime` object. If not present the latest entry's *updated* is used. Treated as UTC if naive datetime.

- `feed_url` – the URL to the feed. Should be the URL that was requested.
- `author` – the author of the feed. Must be either a string (the name) or a dict with name (required) and uri or email (both optional). Can be a list of (may be mixed, too) strings and dicts, too, if there are multiple authors. Required if not every entry has an author element.
- `icon` – an icon for the feed.
- `logo` – a logo for the feed.
- `rights` – copyright information for the feed.
- `rights_type` – the type attribute for the rights element. One of `'html'`, `'text'` or `'xhtml'`. Default is `'text'`.
- `subtitle` – a short description of the feed.
- `subtitle_type` – the type attribute for the subtitle element. One of `'text'`, `'html'`, `'text'` or `'xhtml'`. Default is `'text'`.
- `links` – additional links. Must be a list of dictionaries with href (required) and rel, type, hreflang, title, length (all optional)
- `generator` – the software that generated this feed. This must be a tuple in the form `(name, url, version)`. If you don't want to specify one of them, set the item to *None*.
- `entries` – a list with the entries for the feed. Entries can also be added later with *`add()`*.

For more information on the elements see http://www.atomenabled.org/developers/syndicat

Everywhere where a list is demanded, any iterable can be used.

`add`(**args*, ***kwargs*)
Add a new entry to the feed. This function can either be called with a *`FeedEntry`* or some keyword and positional arguments that are forwarded to the *`FeedEntry`* constructor.

`generate()`
Return a generator that yields pieces of XML.

`get_response()`
Return a response object for the feed.

`to_string()`
Convert the feed into a string.

class `werkzeug.contrib.atom.FeedEntry`(*title=None*, *content=None*, *feed_url=None*, ***kwargs*)
Represents a single entry in a feed.

Parameters

- `title` – the title of the entry. Required.

- `title_type` – the type attribute for the title element. One of `'html'`, `'text'` or `'xhtml'`.
- `content` – the content of the entry.
- `content_type` – the type attribute for the content element. One of `'html'`, `'text'` or `'xhtml'`.
- `summary` – a summary of the entry's content.
- `summary_type` – the type attribute for the summary element. One of `'html'`, `'text'` or `'xhtml'`.
- `url` – the url for the entry.
- `id` – a globally unique id for the entry. Must be an URI. If not present the URL is used, but one of both is required.
- `updated` – the time the entry was modified the last time. Must be a `datetime.datetime` object. Treated as UTC if naive datetime. Required.
- `author` – the author of the entry. Must be either a string (the name) or a dict with name (required) and uri or email (both optional). Can be a list of (may be mixed, too) strings and dicts, too, if there are multiple authors. Required if the feed does not have an author element.
- `published` – the time the entry was initially published. Must be a `datetime.datetime` object. Treated as UTC if naive datetime.
- `rights` – copyright information for the entry.
- `rights_type` – the type attribute for the rights element. One of `'html'`, `'text'` or `'xhtml'`. Default is `'text'`.
- `links` – additional links. Must be a list of dictionaries with href (required) and rel, type, hreflang, title, length (all optional)
- `categories` – categories for the entry. Must be a list of dictionaries with term (required), scheme and label (all optional)
- `xml_base` – The xml base (url) for this feed item. If not provided it will default to the item url.

For more information on the elements see http://www.atomenabled.org/developers/syndicat

Everywhere where a list is demanded, any iterable can be used.

21.2 Sessions

This module contains some helper classes that help one to add session support to a python WSGI application. For full client-side session storage see *securecookie* which implements a secure, client-side session storage.

21.2.1 Application Integration

```
from werkzeug.contrib.sessions import SessionMiddleware, \
     FilesystemSessionStore

app = SessionMiddleware(app, FilesystemSessionStore())
```

The current session will then appear in the WSGI environment as *werkzeug.session*. However it's recommended to not use the middleware but the stores directly in the application. However for very simple scripts a middleware for sessions could be sufficient.

This module does not implement methods or ways to check if a session is expired. That should be done by a cronjob and storage specific. For example to prune unused filesystem sessions one could check the modified time of the files. It sessions are stored in the database the new() method should add an expiration timestamp for the session.

For better flexibility it's recommended to not use the middleware but the store and session object directly in the application dispatching:

```
session_store = FilesystemSessionStore()

def application(environ, start_response):
    request = Request(environ)
    sid = request.cookies.get('cookie_name')
    if sid is None:
        request.session = session_store.new()
    else:
        request.session = session_store.get(sid)
    response = get_the_response_object(request)
    if request.session.should_save:
        session_store.save(request.session)
        response.set_cookie('cookie_name', request.session.sid)
    return response(environ, start_response)
```

21.2.2 Reference

class `werkzeug.contrib.sessions.Session`(*data*, *sid*, *new=False*)

Subclass of a dict that keeps track of direct object changes. Changes in mutable structures are not tracked, for those you have to set *modified* to *True* by hand.

`sid`

The session ID as string.

`new`

True is the cookie was newly created, otherwise *False*

`modified`

Whenever an item on the cookie is set, this attribute is set to *True*. However this does not track modifications inside mutable objects in the session:

```
>>> c = Session({}, sid='deadbeefbabe2c00ffee')
>>> c["foo"] = [1, 2, 3]
>>> c.modified
True
>>> c.modified = False
>>> c["foo"].append(4)
>>> c.modified
False
```

In that situation it has to be set to *modified* by hand so that *should_save* can pick it up.

`should_save`
: True if the session should be saved.

  Changed in version 0.6: By default the session is now only saved if the session is modified, not if it is new like it was before.

class `werkzeug.contrib.sessions.SessionStore`(*session_class=None*)
: Baseclass for all session stores. The Werkzeug contrib module does not implement any useful stores besides the filesystem store, application developers are encouraged to create their own stores.

  Parameters `session_class` – The session class to use. Defaults to *Session*.

  `delete`(*session*)
  : Delete a session.

  `generate_key`(*salt=None*)
  : Simple function that generates a new session key.

  `get`(*sid*)
  : Get a session for this sid or a new session object. This method has to check if the session key is valid and create a new session if that wasn't the case.

  `is_valid_key`(*key*)
  : Check if a key has the correct format.

  `new`()
  : Generate a new session.

  `save`(*session*)
  : Save a session.

  `save_if_modified`(*session*)
  : Save if a session class wants an update.

class `werkzeug.contrib.sessions.FilesystemSessionStore`(*path=None*, *filename_template='werkzeug_%s.sess'*, *session_class=None*, *renew_missing=False*, *mode=420*)

Simple example session store that saves sessions on the filesystem. This store works best on POSIX systems and Windows Vista / Windows Server 2008 and newer.

Changed in version 0.6: *renew_missing* was added. Previously this was considered *True*, now the default changed to *False* and it can be explicitly deactivated.

Parameters

- `path` – the path to the folder used for storing the sessions. If not provided the default temporary directory is used.
- `filename_template` – a string template used to give the session a filename. `%s` is replaced with the session id.
- `session_class` – The session class to use. Defaults to *`Session`*.
- `renew_missing` – set to *True* if you want the store to give the user a new sid if the session was not yet saved.

`list()`

Lists all sessions in the store.

New in version 0.6.

class `werkzeug.contrib.sessions.SessionMiddleware`(*app*, *store*, *cookie_name='session_id'*, *cookie_age=None*, *cookie_expires=None*, *cookie_path='/'*, *cookie_domain=None*, *cookie_secure=None*, *cookie_httponly=False*, *environ_key='werkzeug.session'*)

A simple middleware that puts the session object of a store provided into the WSGI environ. It automatically sets cookies and restores sessions.

However a middleware is not the preferred solution because it won't be as fast as sessions managed by the application itself and will put a key into the WSGI environment only relevant for the application which is against the concept of WSGI.

The cookie parameters are the same as for the `dump_cookie()` function just prefixed with `cookie_`. Additionally *max_age* is called *cookie_age* and not *cookie_max_age* because of backwards compatibility.

21.3 Secure Cookie

This module implements a cookie that is not alterable from the client because it adds a checksum the server checks for. You can use it as session replacement if all you have is a user id or something to mark a logged in user.

Keep in mind that the data is still readable from the client as a normal cookie is. However you don't have to store and flush the sessions you have at the server.

Example usage:

```
>>> from werkzeug.contrib.securecookie import SecureCookie
>>> x = SecureCookie({"foo": 42, "baz": (1, 2, 3)}, "deadbeef")
```

Dumping into a string so that one can store it in a cookie:

```
>>> value = x.serialize()
```

Loading from that string again:

```
>>> x = SecureCookie.unserialize(value, "deadbeef")
>>> x["baz"]
(1, 2, 3)
```

If someone modifies the cookie and the checksum is wrong the unserialize method will fail silently and return a new empty *SecureCookie* object.

Keep in mind that the values will be visible in the cookie so do not store data in a cookie you don't want the user to see.

21.3.1 Application Integration

If you are using the werkzeug request objects you could integrate the secure cookie into your application like this:

```
from werkzeug.utils import cached_property
from werkzeug.wrappers import BaseRequest
from werkzeug.contrib.securecookie import SecureCookie

# don't use this key but a different one; you could just use
# os.urandom(20) to get something random
SECRET_KEY = '\xfa\xdd\xb8z\xae\xe0}4\x8b\xea'

class Request(BaseRequest):

    @cached_property
    def client_session(self):
        data = self.cookies.get('session_data')
        if not data:
            return SecureCookie(secret_key=SECRET_KEY)
        return SecureCookie.unserialize(data, SECRET_KEY)

def application(environ, start_response):
    request = Request(environ, start_response)

    # get a response object here
    response = ...
```

```
    if request.client_session.should_save:
        session_data = request.client_session.serialize()
        response.set_cookie('session_data', session_data,
                            httponly=True)
    return response(environ, start_response)
```

A less verbose integration can be achieved by using shorthand methods:

```
class Request(BaseRequest):

    @cached_property
    def client_session(self):
        return SecureCookie.load_cookie(self, secret_key=COOKIE_SECRET)

def application(environ, start_response):
    request = Request(environ, start_response)

    # get a response object here
    response = ...

    request.client_session.save_cookie(response)
    return response(environ, start_response)
```

21.3.2 Security

The default implementation uses Pickle as this is the only module that used to be available in the standard library when this module was created. If you have simplejson available it's strongly recommended to create a subclass and replace the serialization method:

```
import json
from werkzeug.contrib.securecookie import SecureCookie

class JSONSecureCookie(SecureCookie):
    serialization_method = json
```

The weakness of Pickle is that if someone gains access to the secret key the attacker can not only modify the session but also execute arbitrary code on the server.

21.3.3 Reference

class `werkzeug.contrib.securecookie.SecureCookie`(*data=None*, *secret_key=None*, *new=True*)

Represents a secure cookie. You can subclass this class and provide an alternative mac method. The import thing is that the mac method is a function with a similar interface to the hashlib. Required methods are update() and digest().

Example usage:

```
>>> x = SecureCookie({"foo": 42, "baz": (1, 2, 3)}, "deadbeef")
>>> x["foo"]
42
>>> x["baz"]
(1, 2, 3)
>>> x["blafasel"] = 23
>>> x.should_save
True
```

Parameters

- `data` – the initial data. Either a dict, list of tuples or *None*.
- `secret_key` – the secret key. If not set *None* or not specified it has to be set before *`serialize()`* is called.
- `new` – The initial value of the *new* flag.

`new`
True if the cookie was newly created, otherwise *False*

`modified`
Whenever an item on the cookie is set, this attribute is set to *True*. However this does not track modifications inside mutable objects in the cookie:

```
>>> c = SecureCookie()
>>> c["foo"] = [1, 2, 3]
>>> c.modified
True
>>> c.modified = False
>>> c["foo"].append(4)
>>> c.modified
False
```

In that situation it has to be set to *modified* by hand so that *`should_save`* can pick it up.

static `hash_method()`
The hash method to use. This has to be a module with a new function or a function that creates a hashlib object. Such as *hashlib.md5* Subclasses can override this attribute. The default hash is sha1. Make sure to wrap this in staticmethod() if you store an arbitrary function there such as hashlib.sha1 which might be implemented as a function.

classmethod `load_cookie`(*request*, *key='session'*, *secret_key=None*)
Loads a *`SecureCookie`* from a cookie in request. If the cookie is not set, a new *`SecureCookie`* instanced is returned.

Parameters

- `request` – a request object that has a *cookies* attribute which is a dict of all cookie values.
- `key` – the name of the cookie.

- `secret_key` – the secret key used to unquote the cookie. Always provide the value even though it has no default!

classmethod `quote`(*value*)
Quote the value for the cookie. This can be any object supported by *`serialization_method`*.

Parameters `value` – the value to quote.

`quote_base64` **= True**
if the contents should be base64 quoted. This can be disabled if the serialization process returns cookie safe strings only.

`save_cookie`(*response*, *key='session'*, *expires=None*, *session_expires=None*, *max_age=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=False*, *force=False*)
Saves the SecureCookie in a cookie on response object. All parameters that are not described here are forwarded directly to `set_cookie()`.

Parameters

- `response` – a response object that has a `set_cookie()` method.
- `key` – the name of the cookie.
- `session_expires` – the expiration date of the secure cookie stored information. If this is not provided the cookie *expires* date is used instead.

`serialization_method` **= <module 'pickle' from '/usr/lib/python3.4/pickle.py'>**
the module used for serialization. Unless overriden by subclasses the standard pickle module is used.

`serialize`(*expires=None*)
Serialize the secure cookie into a string.

If expires is provided, the session will be automatically invalidated after expiration when you unseralize it. This provides better protection against session cookie theft.

Parameters `expires` – an optional expiration date for the cookie (a `datetime.datetime` object)

`should_save`
True if the session should be saved. By default this is only true for *`modified`* cookies, not *`new`*.

classmethod `unquote`(*value*)
Unquote the value for the cookie. If unquoting does not work a *`UnquoteError`* is raised.

Parameters `value` – the value to unquote.

classmethod `unserialize`(*string*, *secret_key*)
Load the secure cookie from a serialized string.

Parameters

- `string` – the cookie value to unserialize.
- `secret_key` – the secret key used to serialize the cookie.

Returns a new *SecureCookie*.

exception `werkzeug.contrib.securecookie.UnquoteError`
Internal exception used to signal failures on quoting.

21.4 Cache

The main problem with dynamic Web sites is, well, they're dynamic. Each time a user requests a page, the webserver executes a lot of code, queries the database, renders templates until the visitor gets the page he sees.

This is a lot more expensive than just loading a file from the file system and sending it to the visitor.

For most Web applications, this overhead isn't a big deal but once it becomes, you will be glad to have a cache system in place.

21.4.1 How Caching Works

Caching is pretty simple. Basically you have a cache object lurking around somewhere that is connected to a remote cache or the file system or something else. When the request comes in you check if the current page is already in the cache and if so, you're returning it from the cache. Otherwise you generate the page and put it into the cache. (Or a fragment of the page, you don't have to cache the full thing)

Here is a simple example of how to cache a sidebar for a template:

```
def get_sidebar(user):
    identifier = 'sidebar_for/user%d' % user.id
    value = cache.get(identifier)
    if value is not None:
        return value
    value = generate_sidebar_for(user=user)
    cache.set(identifier, value, timeout=60 * 5)
    return value
```

21.4.2 Creating a Cache Object

To create a cache object you just import the cache system of your choice from the cache module and instantiate it. Then you can start working with that object:

```
>>> from werkzeug.contrib.cache import SimpleCache
>>> c = SimpleCache()
>>> c.set("foo", "value")
>>> c.get("foo")
'value'
>>> c.get("missing") is None
True
```

Please keep in mind that you have to create the cache and put it somewhere you have access to it (either as a module global you can import or you just put it into your WSGI application).

21.4.3 Cache System API

class `werkzeug.contrib.cache.BaseCache`(*default_timeout=300*)

Baseclass for the cache systems. All the cache systems implement this API or a superset of it.

Parameters `default_timeout` – the default timeout (in seconds) that is used if no timeout is specified on *`set()`*. A timeout of 0 indicates that the cache never expires.

`add`(*key*, *value*, *timeout=None*)

Works like *`set()`* but does not overwrite the values of already existing keys.

Parameters

- `key` – the key to set
- `value` – the value for the key
- `timeout` – the cache timeout for the key or the default timeout if not specified. A timeout of 0 indicates that the cache never expires.

Returns Same as *`set()`*, but also `False` for already existing keys.

Return type boolean

`clear`()

Clears the cache. Keep in mind that not all caches support completely clearing the cache. :returns: Whether the cache has been cleared. :rtype: boolean

`dec`(*key*, *delta=1*)

Decrements the value of a key by *delta*. If the key does not yet exist it is initialized with *-delta*.

For supporting caches this is an atomic operation.

Parameters

- `key` – the key to increment.
- `delta` – the delta to subtract.

Returns The new value or *None* for backend errors.

`delete(`*key*`)`
Delete *key* from the cache.

Parameters `key` – the key to delete.

Returns Whether the key existed and has been deleted.

Return type boolean

`delete_many(`**keys*`)`
Deletes multiple keys at once.

Parameters `keys` – The function accepts multiple keys as positional arguments.

Returns Whether all given keys have been deleted.

Return type boolean

`get(`*key*`)`
Look up key in the cache and return the value for it.

Parameters `key` – the key to be looked up.

Returns The value if it exists and is readable, else `None`.

`get_dict(`**keys*`)`
Like *get_many()* but return a dict:

```
d = cache.get_dict("foo", "bar")
foo = d["foo"]
bar = d["bar"]
```

Parameters `keys` – The function accepts multiple keys as positional arguments.

`get_many(`**keys*`)`
Returns a list of values for the given keys. For each key a item in the list is created:

```
foo, bar = cache.get_many("foo", "bar")
```

Has the same error handling as *get()*.

Parameters `keys` – The function accepts multiple keys as positional arguments.

`inc(`*key, delta=1*`)`
Increments the value of a key by *delta*. If the key does not yet exist it is initialized with *delta*.

For supporting caches this is an atomic operation.

Parameters

- `key` – the key to increment.
- `delta` – the delta to add.

Returns The new value or `None` for backend errors.

`set`(*key*, *value*, *timeout=None*)
Add a new key/value to the cache (overwrites value, if key already exists in the cache).

Parameters

- `key` – the key to set
- `value` – the value for the key
- `timeout` – the cache timeout for the key (if not specified, it uses the default timeout). A timeout of 0 idicates that the cache never expires.

Returns `True` if key has been updated, `False` for backend errors. Pickling errors, however, will raise a subclass of `pickle.PickleError`.

Return type boolean

`set_many`(*mapping*, *timeout=None*)
Sets multiple keys and values from a mapping.

Parameters

- `mapping` – a mapping with the keys/values to set.
- `timeout` – the cache timeout for the key (if not specified, it uses the default timeout). A timeout of 0 indicates tht the cache never expires.

Returns Whether all given keys have been set.

Return type boolean

21.4.4 Cache Systems

class `werkzeug.contrib.cache.NullCache`(*default_timeout=300*)
A cache that doesn't cache. This can be useful for unit testing.

Parameters `default_timeout` – a dummy parameter that is ignored but exists for API compatibility with other caches.

class `werkzeug.contrib.cache.SimpleCache`(*threshold=500*, *default_timeout=300*)
Simple memory cache for single process environments. This class exists mainly for the development server and is not 100% thread safe. It tries to use as many atomic operations as possible and no locks for simplicity but it could happen under heavy load that keys are added multiple times.

Parameters

- `threshold` – the maximum number of items the cache stores before it starts deleting some.
- `default_timeout` – the default timeout that is used if no timeout is specified on *`set()`*. A timeout of 0 indicates that the cache never expires.

class `werkzeug.contrib.cache.MemcachedCache`(*servers=None*, *default_timeout=300*, *key_prefix=None*)

A cache that uses memcached as backend.

The first argument can either be an object that resembles the API of a `memcache.Client` or a tuple/list of server addresses. In the event that a tuple/list is passed, Werkzeug tries to import the best available memcache library.

This cache looks into the following packages/modules to find bindings for memcached:

- `pylibmc`
- `google.appengine.api.memcached`
- `memcached`

Implementation notes: This cache backend works around some limitations in memcached to simplify the interface. For example unicode keys are encoded to utf-8 on the fly. Methods such as *`get_dict()`* return the keys in the same format as passed. Furthermore all get methods silently ignore key errors to not cause problems when untrusted user data is passed to the get methods which is often the case in web applications.

Parameters

- `servers` – a list or tuple of server addresses or alternatively a `memcache.Client` or a compatible client.
- `default_timeout` – the default timeout that is used if no timeout is specified on *`set()`*. A timeout of 0 indicates taht the cache never expires.
- `key_prefix` – a prefix that is added before all keys. This makes it possible to use the same memcached server for different applications. Keep in mind that *`clear()`* will also clear keys with a different prefix.

class `werkzeug.contrib.cache.GAEMemcachedCache`

This class is deprecated in favour of *`MemcachedCache`* which now supports Google Appengine as well.

Changed in version 0.8: Deprecated in favour of *`MemcachedCache`*.

class `werkzeug.contrib.cache.RedisCache`(*host='localhost'*, *port=6379*, *password=None*, *db=0*, *default_timeout=300*, *key_prefix=None*, ***kwargs*)

Uses the Redis key-value store as a cache backend.

The first argument can be either a string denoting address of the Redis server or an object resembling an instance of a redis.Redis class.

Note: Python Redis API already takes care of encoding unicode strings on the fly.

New in version 0.7.

New in version 0.8: *key_prefix* was added.

Changed in version 0.8: This cache backend now properly serializes objects.

Changed in version 0.8.3: This cache backend now supports password authentication.

Changed in version 0.10: `**kwargs` is now passed to the redis object.

Parameters

- `host` – address of the Redis server or an object which API is compatible with the official Python Redis client (redis-py).
- `port` – port number on which Redis server listens for connections.
- `password` – password authentication for the Redis server.
- `db` – db (zero-based numeric index) on Redis Server to connect.
- `default_timeout` – the default timeout that is used if no timeout is specified on *`set()`*. A timeout of 0 indicates that the cache never expires.
- `key_prefix` – A prefix that should be added to all keys.

Any additional keyword arguments will be passed to `redis.Redis`.

class `werkzeug.contrib.cache.FileSystemCache`(*cache_dir*, *threshold=500*, *default_timeout=300*, *mode=384*)

A cache that stores the items on the file system. This cache depends on being the only user of the *cache_dir*. Make absolutely sure that nobody but this cache stores files there or otherwise the cache will randomly delete files therein.

Parameters

- `cache_dir` – the directory where cache files are stored.
- `threshold` – the maximum number of items the cache stores before it starts deleting some.

- `default_timeout` – the default timeout that is used if no timeout is specified on *`set()`*. A timeout of 0 indicates that the cache never expires.
- `mode` – the file mode wanted for the cache files, default 0600

21.5 Extra Wrappers

Extra wrappers or mixins contributed by the community. These wrappers can be mixed in into request objects to add extra functionality.

Example:

```
from werkzeug.wrappers import Request as RequestBase
from werkzeug.contrib.wrappers import JSONRequestMixin

class Request(RequestBase, JSONRequestMixin):
    pass
```

Afterwards this request object provides the extra functionality of the *`JSONRequestMixin`*.

class `werkzeug.contrib.wrappers.JSONRequestMixin`
Add json method to a request object. This will parse the input data through simplejson if possible.

`BadRequest` will be raised if the content-type is not json or if the data itself cannot be parsed as json.

`json`
Get the result of simplejson.loads if possible.

class `werkzeug.contrib.wrappers.ProtobufRequestMixin`
Add protobuf parsing method to a request object. This will parse the input data through protobuf if possible.

`BadRequest` will be raised if the content-type is not protobuf or if the data itself cannot be parsed property.

`parse_protobuf`(*proto_type*)
Parse the data into an instance of proto_type.

`protobuf_check_initialization` = **True**
by default the *`ProtobufRequestMixin`* will raise a *`BadRequest`* if the object is not initialized. You can bypass that check by setting this attribute to *False*.

class `werkzeug.contrib.wrappers.RoutingArgsRequestMixin`
This request mixin adds support for the wsgiorg routing args specification.

`routing_args`
The positional URL arguments as *tuple*.

`routing_vars`
The keyword URL arguments as *dict*.

class `werkzeug.contrib.wrappers.ReverseSlashBehaviorRequestMixin`
This mixin reverses the trailing slash behavior of *`script_root`* and *`path`*. This makes it possible to use `urljoin()` directly on the paths.

Because it changes the behavior or `Request` this class has to be mixed in *before* the actual request class:

```
class MyRequest(ReverseSlashBehaviorRequestMixin, Request):
    pass
```

This example shows the differences (for an application mounted on */application* and the request going to */application/foo/bar*):

	normal behavior	reverse behavior
script_root	`/application`	`/application/`
path	`/foo/bar`	`foo/bar`

`path`
Requested path as unicode. This works a bit like the regular path info in the WSGI environment but will not include a leading slash.

`script_root`
The root path of the script includling a trailing slash.

class `werkzeug.contrib.wrappers.DynamicCharsetRequestMixin`
"If this mixin is mixed into a request class it will provide a dynamic *charset* attribute. This means that if the charset is transmitted in the content type headers it's used from there.

Because it changes the behavior or `Request` this class has to be mixed in *before* the actual request class:

```
class MyRequest(DynamicCharsetRequestMixin, Request):
    pass
```

By default the request object assumes that the URL charset is the same as the data charset. If the charset varies on each request based on the transmitted data it's not a good idea to let the URLs change based on that. Most browsers assume either utf-8 or latin1 for the URLs if they have troubles figuring out. It's strongly recommended to set the URL charset to utf-8:

```
class MyRequest(DynamicCharsetRequestMixin, Request):
    url_charset = 'utf-8'
```

New in version 0.6.

`charset`
The charset from the content type.

`default_charset` **= 'latin1'**
the default charset that is assumed if the content type header is missing or does not contain a charset parameter. The default is latin1 which is what

HTTP specifies as default charset. You may however want to set this to utf-8 to better support browsers that do not transmit a charset for incoming data.

`unknown_charset(`*charset*`)`
Called if a charset was provided but is not supported by the Python codecs module. By default latin1 is assumed then to not lose any information, you may override this method to change the behavior.

Parameters `charset` – the charset that was not found.

Returns the replacement charset.

class `werkzeug.contrib.wrappers.DynamicCharsetResponseMixin`
If this mixin is mixed into a response class it will provide a dynamic *charset* attribute. This means that if the charset is looked up and stored in the *Content-Type* header and updates itself automatically. This also means a small performance hit but can be useful if you're working with different charsets on responses.

Because the charset attribute is no a property at class-level, the default value is stored in *default_charset*.

Because it changes the behavior or `Response` this class has to be mixed in *before* the actual response class:

```
class MyResponse(DynamicCharsetResponseMixin, Response):
    pass
```

New in version 0.6.

`charset`
The charset for the response. It's stored inside the Content-Type header as a parameter.

`default_charset` **= 'utf-8'**
the default charset.

21.6 Iter IO

This module implements a *IterIO* that converts an iterator into a stream object and the other way round. Converting streams into iterators requires the greenlet module.

To convert an iterator into a stream all you have to do is to pass it directly to the *IterIO* constructor. In this example we pass it a newly created generator:

```
def foo():
    yield "something\n"
    yield "otherthings"
stream = IterIO(foo())
print stream.read()         # read the whole iterator
```

The other way round works a bit different because we have to ensure that the code execution doesn't take place yet. An *IterIO* call with a callable as first argument does two things. The function itself is passed an *IterIO* stream it can feed. The object returned by the *IterIO* constructor on the other hand is not an stream object but an iterator:

```
def foo(stream):
    stream.write("some")
    stream.write("thing")
    stream.flush()
    stream.write("otherthing")
iterator = IterIO(foo)
print iterator.next()       # prints something
print iterator.next()       # prints otherthing
iterator.next()             # raises StopIteration
```

class `werkzeug.contrib.iterio.IterIO`

Instances of this object implement an interface compatible with the standard Python `file` object. Streams are either read-only or write-only depending on how the object is created.

If the first argument is an iterable a file like object is returned that returns the contents of the iterable. In case the iterable is empty read operations will return the sentinel value.

If the first argument is a callable then the stream object will be created and passed to that function. The caller itself however will not receive a stream but an iterable. The function will be be executed step by step as something iterates over the returned iterable. Each call to `flush()` will create an item for the iterable. If `flush()` is called without any writes in-between the sentinel value will be yielded.

Note for Python 3: due to the incompatible interface of bytes and streams you should set the sentinel value explicitly to an empty bytestring (`b''`) if you are expecting to deal with bytes as otherwise the end of the stream is marked with the wrong sentinel value.

New in version 0.9: *sentinel* parameter was added.

21.7 Fixers

New in version 0.5.

This module includes various helpers that fix bugs in web servers. They may be necessary for some versions of a buggy web server but not others. We try to stay updated with the status of the bugs as good as possible but you have to make sure whether they fix the problem you encounter.

If you notice bugs in webservers not fixed in this module consider contributing a patch.

class `werkzeug.contrib.fixers.CGIRootFix(`*app*, *app_root='/'*`)`
Wrap the application in this middleware if you are using FastCGI or CGI and you have problems with your app root being set to the cgi script's path instead of the path users are going to visit

Changed in version 0.9: Added *app_root* parameter and renamed from *LighttpdCGIRootFix*.

Parameters

- `app` – the WSGI application
- `app_root` – Defaulting to `'/'`, you can set this to something else if your app is mounted somewhere else.

class `werkzeug.contrib.fixers.PathInfoFromRequestUriFix(`*app*`)`
On windows environment variables are limited to the system charset which makes it impossible to store the *PATH_INFO* variable in the environment without loss of information on some systems.

This is for example a problem for CGI scripts on a Windows Apache.

This fixer works by recreating the *PATH_INFO* from *REQUEST_URI*, *REQUEST_URL*, or *UNENCODED_URL* (whatever is available). Thus the fix can only be applied if the webserver supports either of these variables.

Parameters `app` – the WSGI application

class `werkzeug.contrib.fixers.ProxyFix(`*app*, *num_proxies=1*`)`
This middleware can be applied to add HTTP proxy support to an application that was not designed with HTTP proxies in mind. It sets *REMOTE_ADDR*, *HTTP_HOST* from *X-Forwarded* headers. While Werkzeug-based applications already can use *`werkzeug.wsgi.get_host()`* to retrieve the current host even if behind proxy setups, this middleware can be used for applications which access the WSGI environment directly.

If you have more than one proxy server in front of your app, set *num_proxies* accordingly.

Do not use this middleware in non-proxy setups for security reasons.

The original values of *REMOTE_ADDR* and *HTTP_HOST* are stored in the WSGI environment as *werkzeug.proxy_fix.orig_remote_addr* and *werkzeug.proxy_fix.orig_http_host*.

Parameters

- `app` – the WSGI application
- `num_proxies` – the number of proxy servers in front of the app.

`get_remote_addr(`*forwarded_for*`)`
Selects the new remote addr from the given list of ips in X-Forwarded-For. By default it picks the one that the *num_proxies* proxy server provides. Before 0.9 it would always pick the first.

New in version 0.8.

class `werkzeug.contrib.fixers.HeaderRewriterFix`(*app*, *remove_headers=None*, *add_headers=None*)

This middleware can remove response headers and add others. This is for example useful to remove the *Date* header from responses if you are using a server that adds that header, no matter if it's present or not or to add *X-Powered-By* headers:

```
app = HeaderRewriterFix(app, remove_headers=['Date'],
                        add_headers=[('X-Powered-By', 'WSGI')])
```

Parameters

- `app` – the WSGI application
- `remove_headers` – a sequence of header keys that should be removed.
- `add_headers` – a sequence of `(key, value)` tuples that should be added.

class `werkzeug.contrib.fixers.InternetExplorerFix`(*app*, *fix_vary=True*, *fix_attach=True*)

This middleware fixes a couple of bugs with Microsoft Internet Explorer. Currently the following fixes are applied:

- removing of *Vary* headers for unsupported mimetypes which causes troubles with caching. Can be disabled by passing `fix_vary=False` to the constructor. see: http://support.microsoft.com/kb/824847/en-us
- removes offending headers to work around caching bugs in Internet Explorer if *Content-Disposition* is set. Can be disabled by passing `fix_attach=False` to the constructor.

If it does not detect affected Internet Explorer versions it won't touch the request / response.

21.8 WSGI Application Profiler

This module provides a simple WSGI profiler middleware for finding bottlenecks in web application. It uses the `profile` or `cProfile` module to do the profiling and writes the stats to the stream provided (defaults to stderr).

Example usage:

```
from werkzeug.contrib.profiler import ProfilerMiddleware
app = ProfilerMiddleware(app)
```

class `werkzeug.contrib.profiler.MergeStream`(**streams*)

An object that redirects *write* calls to multiple streams. Use this to log to both *sys.stdout* and a file:

```
    f = open('profiler.log', 'w')
    stream = MergeStream(sys.stdout, f)
    profiler = ProfilerMiddleware(app, stream)
```

class werkzeug.contrib.profiler.ProfilerMiddleware(*app*, *stream=None*, *sort_by=('time', 'calls')*, *restrictions=()*, *profile_dir=None*)

Simple profiler middleware. Wraps a WSGI application and profiles a request. This intentionally buffers the response so that timings are more exact.

By giving the *profile_dir* argument, pstat.Stats files are saved to that directory, one file per request. Without it, a summary is printed to *stream* instead.

For the exact meaning of *sort_by* and *restrictions* consult the profile documentation.

New in version 0.9: Added support for *restrictions* and *profile_dir*.

Parameters

- app – the WSGI application to profile.
- stream – the stream for the profiled stats. defaults to stderr.
- sort_by – a tuple of columns to sort the result by.
- restrictions – a tuple of profiling strictions, not used if dumping to *profile_dir*.
- profile_dir – directory name to save pstat files

werkzeug.contrib.profiler.make_action(*app_factory*, *hostname='localhost'*, *port=5000*, *threaded=False*, *processes=1*, *stream=None*, *sort_by=('time', 'calls')*, *restrictions=()*)

Return a new callback for werkzeug.script that starts a local server with the profiler enabled.

```
    from werkzeug.contrib import profiler
    action_profile = profiler.make_action(make_app)
```

21.9 Lint Validation Middleware

Part VI

ADDITIONAL INFORMATION

CHAPTER

TWENTYTWO

IMPORTANT TERMS

This page covers important terms used in the documentation and Werkzeug itself.

22.1 WSGI

WSGI a specification for Python web applications Werkzeug follows. It was specified in the **PEP 333** and is widely supported. Unlike previous solutions it guarantees that web applications, servers and utilties can work together.

22.2 Response Object

For Werkzeug, a response object is an object that works like a WSGI application but does not do any request processing. Usually you have a view function or controller method that processes the request and assambles a response object.

A response object is *not* necessarily the `BaseResponse` object or a subclass thereof.

For example Pylons/webob provide a very similar response class that can be used as well (`webob.Response`).

22.3 View Function

Often people speak of MVC (Model, View, Controller) when developing web applications. However, the Django framework coined MTV (Model, Template, View) which basically means the same but reduces the concept to the data model, a function that processes data from the request and the database and renders a template.

Werkzeug itself does not tell you how you should develop applications, but the documentation often speaks of view functions that work roughly the same. The idea of a view function is that it's called with a request object (and optionally some parameters from an URL rule) and returns a response object.

CHAPTER

TWENTYTHREE

UNICODE

Since early Python 2 days unicode was part of all default Python builds. It allows developers to write applications that deal with non-ASCII characters in a straightforward way. But working with unicode requires a basic knowledge about that matter, especially when working with libraries that do not support it.

Werkzeug uses unicode internally everywhere text data is assumed, even if the HTTP standard is not unicode aware as it. Basically all incoming data is decoded from the charset specified (per default *utf-8*) so that you don't operate on bytestrings any more. Outgoing unicode data is then encoded into the target charset again.

23.1 Unicode in Python

In Python 2 there are two basic string types: *str* and *unicode*. *str* may carry encoded unicode data but it's always represented in bytes whereas the *unicode* type does not contain bytes but charpoints. What does this mean? Imagine you have the German Umlaut *ö*. In ASCII you cannot represent that character, but in the *latin-1* and *utf-8* character sets you can represent it, but they look differently when encoded:

```
>>> u'ö'.encode('latin1')
'\xf6'
>>> u'ö'.encode('utf-8')
'\xc3\xb6'
```

So an *ö* might look totally different depending on the encoding which makes it hard to work with it. The solution is using the *unicode* type (as we did above, note the *u* prefix before the string). The unicode type does not store the bytes for *ö* but the information, that this is a `LATIN SMALL LETTER O WITH DIAERESIS`.

Doing `len(u'ö')` will always give us the expected "1" but `len('ö')` might give different results depending on the encoding of `'ö'`.

23.2 Unicode in HTTP

The problem with unicode is that HTTP does not know what unicode is. HTTP is limited to bytes but this is not a big problem as Werkzeug decodes and encodes for us automatically all incoming and outgoing data. Basically what this means is that data sent from the browser to the web application is per default decoded from an utf-8 bytestring into a *unicode* string. Data sent from the application back to the browser that is not yet a bytestring is then encoded back to utf-8.

Usually this "just works" and we don't have to worry about it, but there are situations where this behavior is problematic. For example the Python 2 IO layer is not unicode aware. This means that whenever you work with data from the file system you have to properly decode it. The correct way to load a text file from the file system looks like this:

```
f = file('/path/to/the_file.txt', 'r')
try:
    text = f.decode('utf-8')    # assuming the file is utf-8 encoded
finally:
    f.close()
```

There is also the codecs module which provides an open function that decodes automatically from the given encoding.

23.3 Error Handling

With Werkzeug 0.3 onwards you can further control the way Werkzeug works with unicode. In the past Werkzeug ignored encoding errors silently on incoming data. This decision was made to avoid internal server errors if the user tampered with the submitted data. However there are situations where you want to abort with a *400 BAD REQUEST* instead of silently ignoring the error.

All the functions that do internal decoding now accept an *errors* keyword argument that behaves like the *errors* parameter of the builtin string method *decode*. The following values are possible:

ignore This is the default behavior and tells the codec to ignore characters that it doesn't understand silently.

replace The codec will replace unknown characters with a replacement character (*U+FFFD* `REPLACEMENT CHARACTER`)

strict Raise an exception if decoding fails.

Unlike the regular python decoding Werkzeug does not raise an `UnicodeDecodeError` if the decoding failed but an *`HTTPUnicodeError`* which is a direct subclass of *UnicodeError* and the *BadRequest* HTTP exception. The reason is that if this exception is not caught by the application but a catch-all for HTTP exceptions exists a default *400 BAD REQUEST* error page is displayed.

There is additional error handling available which is a Werkzeug extension to the regular codec error handling which is called *fallback*. Often you want to use utf-8 but support latin1 as legacy encoding too if decoding failed. For this case you can use the *fallback* error handling. For example you can specify `'fallback:iso-8859-15'` to tell Werkzeug it should try with *iso-8859-15* if *utf-8* failed. If this decoding fails too (which should not happen for most legacy charsets such as *iso-8859-15*) the error is silently ignored as if the error handling was *ignore*.

Further details are available as part of the API documentation of the concrete implementations of the functions or classes working with unicode.

23.4 Request and Response Objects

As request and response objects usually are the central entities of Werkzeug powered applications you can change the default encoding Werkzeug operates on by subclassing these two classes. For example you can easily set the application to utf-7 and strict error handling:

```
from werkzeug.wrappers import BaseRequest, BaseResponse

class Request(BaseRequest):
    charset = 'utf-7'
    encoding_errors = 'strict'

class Response(BaseResponse):
    charset = 'utf-7'
```

Keep in mind that the error handling is only customizable for all decoding but not encoding. If Werkzeug encounters an encoding error it will raise a `UnicodeEncodeError`. It's your responsibility to not create data that is not present in the target charset (a non issue with all unicode encodings such as utf-8).

CHAPTER

TWENTYFOUR

DEALING WITH REQUEST DATA

The most important rule about web development is "Do not trust the user". This is especially true for incoming request data on the input stream. With WSGI this is actually a bit harder than you would expect. Because of that Werkzeug wraps the request stream for you to save you from the most prominent problems with it.

24.1 Missing EOF Marker on Input Stream

The input stream has no end-of-file marker. If you would call the `read()` method on the *wsgi.input* stream you would cause your application to hang on conforming servers. This is actually intentional however painful. Werkzeug solves that problem by wrapping the input stream in a special `LimitedStream`. The input stream is exposed on the request objects as `stream`. This one is either an empty stream (if the form data was parsed) or a limited stream with the contents of the input stream.

24.2 When does Werkzeug Parse?

Werkzeug parses the incoming data under the following situations:

- you access either `form`, `files`, or `stream` and the request method was *POST* or *PUT*.
- if you call `parse_form_data()`.

These calls are not interchangeable. If you invoke `parse_form_data()` you must not use the request object or at least not the attributes that trigger the parsing process.

This is also true if you read from the *wsgi.input* stream before the parsing.

General rule: Leave the WSGI input stream alone. Especially in WSGI middlewares. Use either the parsing functions or the request object. Do not mix multiple WSGI utility libraries for form data parsing or anything else that works on the input stream.

24.3 How does it Parse?

The standard Werkzeug parsing behavior handles three cases:

- input content type was *multipart/form-data*. In this situation the `stream` will be empty and `form` will contain the regular *POST / PUT* data, `files` will contain the uploaded files as `FileStorage` objects.
- input content type was *application/x-www-form-urlencoded*. Then the `stream` will be empty and `form` will contain the regular *POST / PUT* data and `files` will be empty.
- the input content type was neither of them, `stream` points to a `LimitedStream` with the input data for further processing.

Special note on the `get_data` method: Calling this loads the full request data into memory. This is only safe to do if the `max_content_length` is set. Also you can *either* read the stream *or* call `get_data()`.

24.4 Limiting Request Data

To avoid being the victim of a DDOS attack you can set the maximum accepted content length and request field sizes. The `BaseRequest` class has two attributes for that: `max_content_length` and `max_form_memory_size`.

The first one can be used to limit the total content length. For example by setting it to `1024 * 1024 * 16` the request won't accept more than 16MB of transmitted data.

Because certain data can't be moved to the hard disk (regular post data) whereas temporary files can, there is a second limit you can set. The `max_form_memory_size` limits the size of *POST* transmitted form data. By setting it to `1024 * 1024 * 2` you can make sure that all in memory-stored fields is not more than 2MB in size.

This however does *not* affect in-memory stored files if the *stream_factory* used returns a in-memory file.

24.5 How to extend Parsing?

Modern web applications transmit a lot more than multipart form data or url encoded data. Extending the parsing capabilities by subclassing the `BaseRequest` is simple. The following example implements parsing for incoming JSON data:

```
from werkzeug.utils import cached_property
from werkzeug.wrappers import Request
from simplejson import loads

class JSONRequest(Request):
    # accept up to 4MB of transmitted data.
```

```
    max_content_length = 1024 * 1024 * 4

    @cached_property
    def json(self):
        if self.headers.get('content-type') == 'application/json':
            return loads(self.data)
```

CHAPTER

TWENTYFIVE

WERKZEUG CHANGELOG

This file lists all major changes in Werkzeug over the versions. For API breaking changes have a look at *API Changes*, they are listed there in detail.

25.1 Werkzeug Changelog

25.1.1 Version 0.11

(release date and codename to be decided)

- Added `reloader_paths` option to `run_simple` and other functions in `werkzeug.serving`. This allows the user to completely override the Python module watching of Werkzeug with custom paths.
- Many custom cached properties of Werkzeug's classes are now subclasses of Python's `property` type (issue `#616`).
- `bind_to_environ` now doesn't differentiate between implicit and explicit default port numbers in `HTTP_HOST` (pull request `#204`).
- `BuildErrors` are now more informative. They come with a complete sentence as error message, and also provide suggestions (pull request `#691`).
- Fix a bug in the user agent parser where Safari's build number instead of version would be extracted (pull request `#703`).
- Fixed issue where RedisCache set_many was broken for twemproxy, which doesn't support the default MULTI command (pull request `#702`).
- `mimetype` parameters on request and response classes are now always converted to lowercase.
- Changed cache so that cache never expires if timeout is 0. This also fixes an issue with redis setex (issue `#550`)

25.1.2 Version 0.10.5

(bugfix release, release date yet to be decided)

- Reloader: Correctly detect file changes made by moving temporary files over the original, which is e.g. the case with PyCharm (pull request `#722`).

25.1.3 Version 0.10.4

(bugfix release, released on March 26th 2015)

- Re-release of 0.10.3 with packaging artifacts manually removed.

25.1.4 Version 0.10.3

(bugfix release, released on March 26th 2015)

- Re-release of 0.10.2 without packaging artifacts.

25.1.5 Version 0.10.2

(bugfix release, released on March 26th 2015)

- Fixed issue where `empty` could break third-party libraries that relied on keyword arguments (pull request `#675`)
- Improved `Rule.empty` by providing a `'get_empty_kwargs` to allow setting custom kwargs without having to override entire `empty` method. (pull request `#675`)
- Fixed `'extra_files'` parameter for reloader to not cause startup to crash when included in server params
- Using *MultiDict* when building URLs is now not supported again. The behavior introduced several regressions.
- Fix performance problems with stat-reloader (pull request `#715`).

25.1.6 Version 0.10.1

(bugfix release, released on February 3rd 2015)

- Fixed regression with multiple query values for URLs (pull request `#667`).
- Fix issues with eventlet's monkeypatching and the builtin server (pull request `#663`).

25.1.7 Version 0.10

Released on January 30th 2015, codename Bagger.

- Changed the error handling of and improved testsuite for the caches in `contrib.cache`.

- Fixed a bug on Python 3 when creating adhoc ssl contexts, due to *sys.maxint* not being defined.
- Fixed a bug on Python 3, that caused *make_ssl_devcert()* to fail with an exception.
- Added exceptions for 504 and 505.
- Added support for ChromeOS detection.
- Added UUID converter to the routing system.
- Added message that explains how to quit the server.
- Fixed a bug on Python 2, that caused `len` for *werkzeug.datastructures.CombinedMultiDict* to crash.
- Added support for stdlib pbkdf2 hmac if a compatible digest is found.
- Ported testsuite to use `py.test`.
- Minor optimizations to various middlewares (pull requests `#496` and `#571`).
- Use stdlib `ssl` module instead of `OpenSSL` for the builtin server (issue `#434`). This means that OpenSSL contexts are not supported anymore, but instead `ssl.SSLContext` from the stdlib.
- Allow protocol-relative URLs when building external URLs.
- Fixed Atom syndication to print time zone offset for tz-aware datetime objects (pull request `#254`).
- Improved reloader to track added files and to recover from broken sys.modules setups with syntax errors in packages.
- `cache.RedisCache` now supports arbitrary `**kwargs` for the redis object.
- `werkzeug.test.Client` now uses the original request method when resolving 307 redirects (pull request `#556`).
- `werkzeug.datastructures.MIMEAccept` now properly deals with mimetype parameters (pull request `#205`).
- `werkzeug.datastructures.Accept` now handles a quality of `0` as intolerable, as per RFC 2616 (pull request `#536`).
- `werkzeug.urls.url_fix` now properly encodes hostnames with `idna` encoding (issue `#559`). It also doesn't crash on malformed URLs anymore (issue `#582`).
- `werkzeug.routing.MapAdapter.match` now recognizes the difference between the path `/` and an empty one (issue `#360`).
- The interactive debugger now tries to decode non-ascii filenames (issue `#469`).
- Increased default key size of generated SSL certificates to 1024 bits (issue `#611`).
- Added support for specifying a `Response` subclass to use when calling *redirect()*.

- `werkzeug.test.EnvironBuilder` now doesn't use the request method anymore to guess the content type, and purely relies on the `form`, `files` and `input_stream` properties (issue #620).
- Added Symbian to the user agent platform list.
- Fixed make_conditional to respect automatically_set_content_length
- Unset `Content-Length` when writing to response.stream (issue #451)
- `wrappers.Request.method` is now always uppercase, eliminating inconsistencies of the WSGI environment (issue 647).
- `routing.Rule.empty` now works correctly with subclasses of `Rule` (pull request #645).
- Made map updating safe in light of concurrent updates.
- Allow multiple values for the same field for url building (issue #658).

25.1.8 Version 0.9.7

(bugfix release, release date to be decided)

- Fix unicode problems in `werkzeug.debug.tbtools`.
- Fix Python 3-compatibility problems in `werkzeug.posixemulation`.
- Backport fix of fatal typo for `ImmutableList` (issue #492).
- Make creation of the cache dir for `FileSystemCache` atomic (issue #468).
- Use native strings for memcached keys to work with Python 3 client (issue #539).
- Fix charset detection for `werkzeug.debug.tbtools.Frame` objects (issues #547 and #532).
- Fix `AttributeError` masking in `werkzeug.utils.import_string` (issue #182).
- Explicitly shut down server (issue #519).
- Fix timeouts greater than 2592000 being misinterpreted as UNIX timestamps in `werkzeug.contrib.cache.MemcachedCache` (issue #533).
- Fix bug where `werkzeug.exceptions.abort` would raise an arbitrary subclass of the expected class (issue #422).
- Fix broken `jsrouting` (due to removal of `werkzeug.templates`)
- `werkzeug.urls.url_fix` now doesn't crash on malformed URLs anymore, but returns them unmodified. This is a cheap workaround for #582, the proper fix is included in version 0.10.
- The repr of `werkzeug.wrappers.Request` doesn't crash on non-ASCII-values anymore (pull request #466).
- Fix bug in `cache.RedisCache` when combined with `redis.StrictRedis` object (pull request #583).

- The `qop` parameter for `WWW-Authenticate` headers is now always quoted, as required by RFC 2617 (issue `#633`).
- Fix bug in `werkzeug.contrib.cache.SimpleCache` with Python 3 where add/set may throw an exception when pruning old entries from the cache (pull request `#651`).

25.1.9 Version 0.9.6

(bugfix release, released on June 7th 2014)

- Added a safe conversion for IRI to URI conversion and use that internally to work around issues with spec violations for protocols such as `itms-service`.

25.1.10 Version 0.9.5

(bugfix release, released on June 7th 2014)

- Forward charset argument from request objects to the environ builder.
- Fixed error handling for missing boundaries in multipart data.
- Fixed session creation on systems without `os.urandom()`.
- Fixed pluses in dictionary keys not being properly URL encoded.
- Fixed a problem with deepcopy not working for multi dicts.
- Fixed a double quoting issue on redirects.
- Fixed a problem with unicode keys appearing in headers on 2.x.
- Fixed a bug with unicode strings in the test builder.
- Fixed a unicode bug on Python 3 in the WSGI profiler.
- Fixed an issue with the safe string compare function on Python 2.7.7 and Python 3.4.

25.1.11 Version 0.9.4

(bugfix release, released on August 26th 2013)

- Fixed an issue with Python 3.3 and an edge case in cookie parsing.
- Fixed decoding errors not handled properly through the WSGI decoding dance.
- Fixed URI to IRI conversion incorrectly decoding percent signs.

25.1.12 Version 0.9.3

(bugfix release, released on July 25th 2013)

- Restored behavior of the `data` descriptor of the request class to pre 0.9 behavior. This now also means that `.data` and `.get_data()` have different behavior. New code should use `.get_data()` always.

  In addition to that there is now a flag for the `.get_data()` method that controls what should happen with form data parsing and the form parser will honor cached data. This makes dealing with custom form data more consistent.

25.1.13 Version 0.9.2

(bugfix release, released on July 18th 2013)

- Added *unsafe* parameter to *url_quote()*.
- Fixed an issue with *url_quote_plus()* not quoting '+' correctly.
- Ported remaining parts of `RedisCache` to Python 3.3.
- Ported remaining parts of `MemcachedCache` to Python 3.3
- Fixed a deprecation warning in the contrib atom module.
- Fixed a regression with setting of content types through the headers dictionary instead with the content type parameter.
- Use correct name for stdlib secure string comparision function.
- Fixed a wrong reference in the docstring of *release_local()*.
- Fixed an *AttributeError* that sometimes occurred when accessing the *werkzeug.wrappers.BaseResponse.is_streamed* attribute.

25.1.14 Version 0.9.1

(bugfix release, released on June 14th 2013)

- Fixed an issue with integers no longer being accepted in certain parts of the routing system or URL quoting functions.
- Fixed an issue with *url_quote* not producing the right escape codes for single digit codepoints.
- Fixed an issue with *SharedDataMiddleware* not reading the path correctly and breaking on etag generation in some cases.
- Properly handle *Expect: 100-continue* in the development server to resolve issues with curl.
- Automatically exhaust the input stream on request close. This should fix issues where not touching request files results in a timeout.

- Fixed exhausting of streams not doing anything if a non-limited stream was passed into the multipart parser.
- Raised the buffer sizes for the multipart parser.

25.1.15 Version 0.9

Released on June 13nd 2013, codename Planierraupe.

- Added support for `tell()` on the limited stream.
- `ETags` now is nonzero if it contains at least one etag of any kind, including weak ones.
- Added a workaround for a bug in the stdlib for SSL servers.
- Improved SSL interface of the devserver so that it can generate certificates easily and load them from files.
- Refactored test client to invoke the open method on the class for redirects. This makes subclassing more powerful.
- `werkzeug.wsgi.make_chunk_iter()` and `werkzeug.wsgi.make_line_iter()` now support processing of iterators and streams.
- URL generation by the routing system now no longer quotes +.
- URL fixing now no longer quotes certain reserved characters.
- The `werkzeug.security.generate_password_hash()` and check functions now support any of the hashlib algorithms.
- *wsgi.get_current_url* is now ascii safe for browsers sending non-ascii data in query strings.
- improved parsing behavior for `werkzeug.http.parse_options_header()`
- added more operators to local proxies.
- added a hook to override the default converter in the routing system.
- The description field of HTTP exceptions is now always escaped. Use markup objects to disable that.
- Added number of proxy argument to the proxy fix to make it more secure out of the box on common proxy setups. It will by default no longer trust the x-forwarded-for header as much as it did before.
- Added support for fragment handling in URI/IRI functions.
- Added custom class support for `werkzeug.http.parse_dict_header()`.
- Renamed *LighttpdCGIRootFix* to *CGIRootFix*.
- Always treat + as safe when fixing URLs as people love misusing them.
- Added support to profiling into directories in the contrib profiler.

- The escape function now by default escapes quotes.
- Changed repr of exceptions to be less magical.
- Simplified exception interface to no longer require environmnts to be passed to recieve the response object.
- Added sentinel argument to IterIO objects.
- Added pbkdf2 support for the security module.
- Added a plain request type that disables all form parsing to only leave the stream behind.
- Removed support for deprecated *fix_headers*.
- Removed support for deprecated *header_list*.
- Removed support for deprecated parameter for *iter_encoded*.
- Removed support for deprecated non-silent usage of the limited stream object.
- Removed support for previous dummy *writable* parameter on the cached property.
- Added support for explicitly closing request objects to close associated resources.
- Conditional request handling or access to the data property on responses no longer ignores direct passthrough mode.
- Removed werkzeug.templates and werkzeug.contrib.kickstart.
- Changed host lookup logic for forwarded hosts to allow lists of hosts in which case only the first one is picked up.
- Added *wsgi.get_query_string*, *wsgi.get_path_info* and *wsgi.get_script_name* and made the *wsgi.pop_path_info* and *wsgi.peek_path_info* functions perform unicode decoding. This was necessary to avoid having to expose the WSGI encoding dance on Python 3.
- Added *content_encoding* and *content_md5* to the request object's common request descriptor mixin.
- added *options* and *trace* to the test client.
- Overhauled the utilization of the input stream to be easier to use and better to extend. The detection of content payload on the input side is now more compliant with HTTP by detecting off the content type header instead of the request method. This also now means that the stream property on the request class is always available instead of just when the parsing fails.
- Added support for using `werkzeug.wrappers.BaseResponse` in a with statement.
- Changed *get_app_iter* to fetch the response early so that it does not fail when wrapping a response iterable. This makes filtering easier.
- Introduced *get_data* and *set_data* methods for responses.

- Introduced *get_data* for requests.
- Soft deprecated the *data* descriptors for request and response objects.
- Added *as_bytes* operations to some of the headers to simplify working with things like cookies.
- Made the debugger paste tracebacks into github's gist service as private pastes.

25.1.16 Version 0.8.4

(bugfix release, release date to be announced)

- Added a favicon to the debugger which fixes problem with state changes being triggered through a request to /favicon.ico in Google Chrome. This should fix some problems with Flask and other frameworks that use context local objects on a stack with context preservation on errors.
- Fixed an issue with scolling up in the debugger.
- Fixed an issue with debuggers running on a different URL than the URL root.
- Fixed a problem with proxies not forwarding some rarely used special methods properly.
- Added a workaround to prevent the XSS protection from Chrome breaking the debugger.
- Skip redis tests if redis is not running.
- Fixed a typo in the multipart parser that caused content-type to not be picked up properly.

25.1.17 Version 0.8.3

(bugfix release, released on February 5th 2012)

- Fixed another issue with `werkzeug.wsgi.make_line_iter()` where lines longer than the buffer size were not handled properly.
- Restore stdout after debug console finished executing so that the debugger can be used on GAE better.
- Fixed a bug with the redis cache for int subclasses (affects bool caching).
- Fixed an XSS problem with redirect targets coming from untrusted sources.
- Redis cache backend now supports password authentication.

25.1.18 Version 0.8.2

(bugfix release, released on December 16th 2011)

- Fixed a problem with request handling of the builtin server not responding to socket errors properly.
- The routing request redirect exception's code attribute is now used properly.
- Fixed a bug with shutdowns on Windows.
- Fixed a few unicode issues with non-ascii characters being hardcoded in URL rules.
- Fixed two property docstrings being assigned to fdel instead of `__doc__`.
- Fixed an issue where CRLF line endings could be split into two by the line iter function, causing problems with multipart file uploads.

25.1.19 Version 0.8.1

(bugfix release, released on September 30th 2011)

- Fixed an issue with the memcache not working properly.
- Fixed an issue for Python 2.7.1 and higher that broke copying of multidicts with `copy.copy()`.
- Changed hashing methodology of immutable ordered multi dicts for a potential problem with alternative Python implementations.

25.1.20 Version 0.8

Released on September 29th 2011, codename Lötkolben

- Removed data structure specific KeyErrors for a general purpose `BadRequestKeyError`.
- Documented `werkzeug.wrappers.BaseRequest._load_form_data()`.
- The routing system now also accepts strings instead of dictionaries for the *query_args* parameter since we're only passing them through for redirects.
- Werkzeug now automatically sets the content length immediately when the *data* attribute is set for efficiency and simplicity reasons.
- The routing system will now normalize server names to lowercase.
- The routing system will no longer raise ValueErrors in case the configuration for the server name was incorrect. This should make deployment much easier because you can ignore that factor now.
- Fixed a bug with parsing HTTP digest headers. It rejected headers with missing nc and nonce params.
- Proxy fix now also updates wsgi.url_scheme based on X-Forwarded-Proto.
- Added support for key prefixes to the redis cache.

- Added the ability to suppress some auto corrections in the wrappers that are now controlled via *autocorrect_location_header* and *automatically_set_content_length* on the response objects.
- Werkzeug now uses a new method to check that the length of incoming data is complete and will raise IO errors by itself if the server fails to do so.
- `make_line_iter()` now requires a limit that is not higher than the length the stream can provide.
- Refactored form parsing into a form parser class that makes it possible to hook into individual parts of the parsing process for debugging and extending.
- For conditional responses the content length is no longer set when it is already there and added if missing.
- Immutable datastructures are hashable now.
- Headers datastructure no longer allows newlines in values to avoid header injection attacks.
- Made it possible through subclassing to select a different remote addr in the proxy fix.
- Added stream based URL decoding. This reduces memory usage on large transmitted form data that is URL decoded since Werkzeug will no longer load all the unparsed data into memory.
- Memcache client now no longer uses the buggy cmemcache module and supports pylibmc. GAE is not tried automatically and the dedicated class is no longer necessary.
- Redis cache now properly serializes data.
- Removed support for Python 2.4

25.1.21 Version 0.7.2

(bugfix release, released on September 30th 2011)

- Fixed a CSRF problem with the debugger.
- The debugger is now generating private pastes on lodgeit.
- If URL maps are now bound to environments the query arguments are properly decoded from it for redirects.

25.1.22 Version 0.7.1

(bugfix release, released on July 26th 2011)

- Fixed a problem with newer versions of IPython.
- Disabled pyinotify based reloader which does not work reliably.

25.1.23 Version 0.7

Released on July 24th 2011, codename Schraubschlüssel

- Add support for python-libmemcached to the Werkzeug cache abstraction layer.
- Improved `url_decode()` and `url_encode()` performance.
- Fixed an issue where the SharedDataMiddleware could cause an internal server error on weird paths when loading via pkg_resources.
- Fixed an URL generation bug that caused URLs to be invalid if a generated component contains a colon.
- `werkzeug.import_string()` now works with partially set up packages properly.
- Disabled automatic socket switching for IPv6 on the development server due to problems it caused.
- Werkzeug no longer overrides the Date header when creating a conditional HTTP response.
- The routing system provides a method to retrieve the matching methods for a given path.
- The routing system now accepts a parameter to change the encoding error behaviour.
- The local manager can now accept custom ident functions in the constructor that are forwarded to the wrapped local objects.
- url_unquote_plus now accepts unicode strings again.
- Fixed an issue with the filesystem session support's prune function and concurrent usage.
- Fixed a problem with external URL generation discarding the port.
- Added support for pylibmc to the Werkzeug cache abstraction layer.
- Fixed an issue with the new multipart parser that happened when a linebreak happened to be on the chunk limit.
- Cookies are now set properly if ports are in use. A runtime error is raised if one tries to set a cookie for a domain without a dot.
- Fixed an issue with Template.from_file not working for file descriptors.
- Reloader can now use inotify to track reloads. This requires the pyinotify library to be installed.
- Werkzeug debugger can now submit to custom lodgeit installations.
- redirect function's status code assertion now allows 201 to be used as redirection code. While it's not a real redirect, it shares enough with redirects for the function to still be useful.
- Fixed securecookie for pypy.

- Fixed *ValueErrors* being raised on calls to *best_match* on *MIMEAccept* objects when invalid user data was supplied.
- Deprecated *werkzeug.contrib.kickstart* and *werkzeug.contrib.testtools*
- URL routing now can be passed the URL arguments to keep them for redirects. In the future matching on URL arguments might also be possible.
- Header encoding changed from utf-8 to latin1 to support a port to Python 3. Bytestrings passed to the object stay untouched which makes it possible to have utf-8 cookies. This is a part where the Python 3 version will later change in that it will always operate on latin1 values.
- Fixed a bug in the form parser that caused the last character to be dropped off if certain values in multipart data are used.
- Multipart parser now looks at the part-individual content type header to override the global charset.
- Introduced mimetype and mimetype_params attribute for the file storage object.
- Changed FileStorage filename fallback logic to skip special filenames that Python uses for marking special files like stdin.
- Introduced more HTTP exception classes.
- *call_on_close* now can be used as a decorator.
- Support for redis as cache backend.
- Added *BaseRequest.scheme*.
- Support for the RFC 5789 PATCH method.
- New custom routing parser and better ordering.
- Removed support for *is_behind_proxy*. Use a WSGI middleware instead that rewrites the *REMOTE_ADDR* according to your setup. Also see the *werkzeug.contrib.fixers.ProxyFix* for a drop-in replacement.
- Added cookie forging support to the test client.
- Added support for host based matching in the routing system.
- Switched from the default 'ignore' to the better 'replace' unicode error handling mode.
- The builtin server now adds a function named 'werkzeug.server.shutdown' into the WSGI env to initiate a shutdown. This currently only works in Python 2.6 and later.
- Headers are now assumed to be latin1 for better compatibility with Python 3 once we have support.
- Added *werkzeug.security.safe_join()*.
- Added *accept_json* property analogous to *accept_html* on the *werkzeug.datastructures.MIMEAccept*.

- *werkzeug.utils.import_string()* now fails with much better error messages that pinpoint to the problem.
- Added support for parsing of the *If-Range* header (*werkzeug.http.parse_if_range_header()* and *werkzeug.datastructures.IfRange*).
- Added support for parsing of the *Range* header (*werkzeug.http.parse_range_header()* and *werkzeug.datastructures.Range*).
- Added support for parsing of the *Content-Range* header of responses and provided an accessor object for it (*werkzeug.http.parse_content_range_header()* and *werkzeug.datastructures.ContentRange*).

25.1.24 Version 0.6.2

(bugfix release, released on April 23th 2010)

- renamed the attribute *implicit_seqence_conversion* attribute of the request object to *implicit_sequence_conversion*.

25.1.25 Version 0.6.1

(bugfix release, released on April 13th 2010)

- heavily improved local objects. Should pick up standalone greenlet builds now and support proxies to free callables as well. There is also a stacked local now that makes it possible to invoke the same application from within itself by pushing current request/response on top of the stack.
- routing build method will also build non-default method rules properly if no method is provided.
- added proper IPv6 support for the builtin server.
- windows specific filesystem session store fixes. (should now be more stable under high concurrency)
- fixed a *NameError* in the session system.
- fixed a bug with empty arguments in the werkzeug.script system.
- fixed a bug where log lines will be duplicated if an application uses `logging.basicConfig()` (#499)
- added secure password hashing and checking functions.
- *HEAD* is now implicitly added as method in the routing system if *GET* is present. Not doing that was considered a bug because often code assumed that this is the case and in web servers that do not normalize *HEAD* to *GET* this could break *HEAD* requests.
- the script support can start SSL servers now.

25.1.26 Version 0.6

Released on Feb 19th 2010, codename Hammer.

- removed pending deprecations
- sys.path is now printed from the testapp.
- fixed an RFC 2068 incompatibility with cookie value quoting.
- the `FileStorage` now gives access to the multipart headers.
- *cached_property.writeable* has been deprecated.
- `MapAdapter.match()` now accepts a *return_rule* keyword argument that returns the matched *Rule* instead of just the *endpoint*
- *routing.Map.bind_to_environ()* raises a more correct error message now if the map was bound to an invalid WSGI environment.
- added support for SSL to the builtin development server.
- Response objects are no longer modified in place when they are evaluated as WSGI applications. For backwards compatibility the *fix_headers* function is still called in case it was overridden. You should however change your application to use *get_wsgi_headers* if you need header modifications before responses are sent as the backwards compatibility support will go away in future versions.
- `append_slash_redirect()` no longer requires the QUERY_STRING to be in the WSGI environment.
- added *DynamicCharsetResponseMixin*
- added *DynamicCharsetRequestMixin*
- added `BaseRequest.url_charset`
- request and response objects have a default *__repr__* now.
- builtin data structures can be pickled now.
- the form data parser will now look at the filename instead the content type to figure out if it should treat the upload as regular form data or file upload. This fixes a bug with Google Chrome.
- improved performance of *make_line_iter* and the multipart parser for binary uploads.
- fixed `is_streamed`
- fixed a path quoting bug in *EnvironBuilder* that caused PATH_INFO and SCRIPT_NAME to end up in the environ unquoted.
- `werkzeug.BaseResponse.freeze()` now sets the content length.
- for unknown HTTP methods the request stream is now always limited instead of being empty. This makes it easier to implement DAV and other protocols on top of Werkzeug.

- added `werkzeug.MIMEAccept.best_match()`
- multi-value test-client posts from a standard dictionary are now supported. Previously you had to use a multi dict.
- rule templates properly work with submounts, subdomains and other rule factories now.
- deprecated non-silent usage of the `werkzeug.LimitedStream`.
- added support for IRI handling to many parts of Werkzeug.
- development server properly logs to the werkzeug logger now.
- added `werkzeug.extract_path_info()`
- fixed a querystring quoting bug in `url_fix()`
- added *fallback_mimetype* to `werkzeug.SharedDataMiddleware`.
- deprecated `BaseResponse.iter_encoded()`'s charset parameter.
- added `BaseResponse.make_sequence()`, `BaseResponse.is_sequence` and `BaseResponse._ensure_sequence()`.
- added better __repr__ of `werkzeug.Map`
- *import_string* accepts unicode strings as well now.
- development server doesn't break on double slashes after the host name.
- better *__repr__* and *__str__* of *werkzeug.exceptions.HTTPException*
- test client works correctly with multiple cookies now.
- the *werkzeug.routing.Map* now has a class attribute with the default converter mapping. This helps subclasses to override the converters without passing them to the constructor.
- implemented `OrderedMultiDict`
- improved the session support for more efficient session storing on the filesystem. Also added support for listing of sessions currently stored in the filesystem session store.
- werkzeug no longer utilizes the Python time module for parsing which means that dates in a broader range can be parsed.
- the wrappers have no class attributes that make it possible to swap out the dict and list types it uses.
- werkzeug debugger should work on the appengine dev server now.
- the URL builder supports dropping of unexpected arguments now. Previously they were always appended to the URL as query string.
- profiler now writes to the correct stream.

25.1.27 Version 0.5.1

(bugfix release for 0.5, released on July 9th 2009)

- fixed boolean check of `FileStorage`
- url routing system properly supports unicode URL rules now.
- file upload streams no longer have to provide a truncate() method.
- implemented `BaseRequest._form_parsing_failed()`.
- fixed #394
- `ImmutableDict.copy()`, `ImmutableMultiDict.copy()` and `ImmutableTypeConversionDict.copy()` return mutable shallow copies.
- fixed a bug with the *make_runserver* script action.
- `MultiDict.items()` and `MutiDict.iteritems()` now accept an argument to return a pair for each value of each key.
- the multipart parser works better with hand-crafted multipart requests now that have extra newlines added. This fixes a bug with setuptools uploades not handled properly (#390)
- fixed some minor bugs in the atom feed generator.
- fixed a bug with client cookie header parsing being case sensitive.
- fixed a not-working deprecation warning.
- fixed package loading for `SharedDataMiddleware`.
- fixed a bug in the secure cookie that made server-side expiration on servers with a local time that was not set to UTC impossible.
- fixed console of the interactive debugger.

25.1.28 Version 0.5

Released on April 24th, codename Schlagbohrer.

- requires Python 2.4 now
- fixed a bug in `IterIO`
- added `MIMEAccept` and `CharsetAccept` that work like the regular `Accept` but have extra special normalization for mimetypes and charsets and extra convenience methods.
- switched the serving system from wsgiref to something homebrew.
- the `Client` now supports cookies.
- added the *`fixers`* module with various fixes for webserver bugs and hosting setup side-effects.

- added *werkzeug.contrib.wrappers*
- added `is_hop_by_hop_header()`
- added `is_entity_header()`
- added `remove_hop_by_hop_headers()`
- added `pop_path_info()`
- added `peek_path_info()`
- added `wrap_file()` and `FileWrapper`
- moved *LimitedStream* from the contrib package into the regular werkzeug one and changed the default behavior to raise exceptions rather than stopping without warning. The old class will stick in the module until 0.6.
- implemented experimental multipart parser that replaces the old CGI hack.
- added `dump_options_header()` and `parse_options_header()`
- added `quote_header_value()` and `unquote_header_value()`
- `url_encode()` and `url_decode()` now accept a separator argument to switch between *&* and *;* as pair separator. The magic switch is no longer in place.
- all form data parsing functions as well as the `BaseRequest` object have parameters (or attributes) to limit the number of incoming bytes (either totally or per field).
- added `LanguageAccept`
- request objects are now enforced to be read only for all collections.
- added many new collection classes, refactored collections in general.
- test support was refactored, semi-undocumented *werkzeug.test.File* was replaced by `werkzeug.FileStorage`.
- `EnvironBuilder` was added and unifies the previous distinct `create_environ()`, `Client` and `BaseRequest.from_values()`. They all work the same now which is less confusing.
- officially documented imports from the internal modules as undefined behavior. These modules were never exposed as public interfaces.
- removed *FileStorage.__len__* which previously made the object falsy for browsers not sending the content length which all browsers do.
- `SharedDataMiddleware` uses *wrap_file* now and has a configurable cache timeout.
- added `CommonRequestDescriptorsMixin`
- added `CommonResponseDescriptorsMixin.mimetype_params`
- added `werkzeug.contrib.lint`
- added *passthrough_errors* to *run_simple*.

- added *secure_filename*
- added `make_line_iter()`
- `MultiDict` copies now instead of revealing internal lists to the caller for *getlist* and iteration functions that return lists.
- added `follow_redirect` to the `open()` of `Client`.
- added support for *extra_files* in `make_runserver()`

25.1.29 Version 0.4.1

(Bugfix release, released on January 11th 2009)

- *werkzeug.contrib.cache.Memcached* accepts now objects that implement the memcache.Client interface as alternative to a list of strings with server addresses. There is also now a *GAEMemcachedCache* that connects to the Google appengine cache.
- explicitly convert secret keys to bytestrings now because Python 2.6 no longer does that.
- *url_encode* and all interfaces that call it, support ordering of options now which however is disabled by default.
- the development server no longer resolves the addresses of clients.
- Fixed a typo in *werkzeug.test* that broke *File*.
- *Map.bind_to_environ* uses the *Host* header now if available.
- Fixed *BaseCache.get_dict* (#345)
- *werkzeug.test.Client* can now run the application buffered in which case the application is properly closed automatically.
- Fixed *Headers.set* (#354). Caused header duplication before.
- Fixed *Headers.pop* (#349). default parameter was not properly handled.
- Fixed UnboundLocalError in *create_environ* (#351)
- *Headers* is more compatible with wsgiref now.
- *Template.render* accepts multidicts now.
- dropped support for Python 2.3

25.1.30 Version 0.4

Released on November 23rd 2008, codename Schraubenzieher.

- *Client* supports an empty *data* argument now.

- fixed a bug in *Response.application* that made it impossible to use it as method decorator.
- the session system should work on appengine now
- the secure cookie works properly in load balanced environments with different cpu architectures now.
- *CacheControl.no_cache* and *CacheControl.private* behavior changed to reflect the possibilities of the HTTP RFC. Setting these attributes to *None* or *True* now sets the value to "the empty value". More details in the documentation.
- fixed *werkzeug.contrib.atom.AtomFeed.__call__*. (#338)
- *BaseResponse.make_conditional* now always returns *self*. Previously it didn't for post requests and such.
- fixed a bug in boolean attribute handling of *html* and *xhtml*.
- added graceful error handling to the debugger pastebin feature.
- added a more list like interface to *Headers* (slicing and indexing works now)
- fixed a bug with the *__setitem__* method of *Headers* that didn't properly remove all keys on replacing.
- added *remove_entity_headers* which removes all entity headers from a list of headers (or a *Headers* object)
- the responses now automatically call *remove_entity_headers* if the status code is 304.
- fixed a bug with *Href* query parameter handling. Previously the last item of a call to *Href* was not handled properly if it was a dict.
- headers now support a *pop* operation to better work with environ properties.

25.1.31 Version 0.3.1

(bugfix release, released on June 24th 2008)

- fixed a security problem with *werkzeug.contrib.SecureCookie*. More details available in the release announcement.

25.1.32 Version 0.3

Released on June 14th 2008, codename EUR325CAT6.

- added support for redirecting in url routing.
- added *Authorization* and *AuthorizationMixin*
- added *WWWAuthenticate* and *WWWAuthenticateMixin*
- added *parse_list_header*

- added *parse_dict_header*
- added *parse_authorization_header*
- added *parse_www_authenticate_header*
- added *_get_current_object* method to *LocalProxy* objects
- added *parse_form_data*
- *MultiDict*, *CombinedMultiDict*, *Headers*, and *EnvironHeaders* raise special key errors now that are subclasses of *BadRequest* so if you don't catch them they give meaningful HTTP responses.
- added support for alternative encoding error handling and the new *HTTPUnicodeError* which (if not caught) behaves like a *BadRequest*.
- added *BadRequest.wrap*.
- added ETag support to the SharedDataMiddleware and added an option to disable caching.
- fixed *is_xhr* on the request objects.
- fixed error handling of the url adapter's *dispatch* method. (#318)
- fixed bug with *SharedDataMiddleware*.
- fixed *Accept.values*.
- *EnvironHeaders* contain content-type and content-length now
- *url_encode* treats lists and tuples in dicts passed to it as multiple values for the same key so that one doesn't have to pass a *MultiDict* to the function.
- added *validate_arguments*
- added *BaseRequest.application*
- improved Python 2.3 support
- *run_simple* accepts *use_debugger* and *use_evalex* parameters now, like the *make_runserver* factory function from the script module.
- the *environ_property* is now read-only by default
- it's now possible to initialize requests as "shallow" requests which causes runtime errors if the request object tries to consume the input stream.

25.1.33 Version 0.2

Released Feb 14th 2008, codename Faustkeil.

- Added *AnyConverter* to the routing system.
- Added *werkzeug.contrib.securecookie*
- Exceptions have a `get_response()` method that return a response object

- fixed the path ordering bug (#293), thanks Thomas Johansson
- *BaseReporterStream* is now part of the werkzeug contrib module. From Werkzeug 0.3 onwards you will have to import it from there.
- added *DispatcherMiddleware*.
- *RequestRedirect* is now a subclass of *HTTPException* and uses a 301 status code instead of 302.
- *url_encode* and *url_decode* can optionally treat keys as unicode strings now, too.
- *werkzeug.script* has a different caller format for boolean arguments now.
- renamed *lazy_property* to *cached_property*.
- added *import_string*.
- added is_* properties to request objects.
- added *empty()* method to routing rules.
- added *werkzeug.contrib.profiler*.
- added *extends* to *Headers*.
- added *dump_cookie* and *parse_cookie*.
- added *as_tuple* to the *Client*.
- added *werkzeug.contrib.testtools*.
- added *werkzeug.unescape*
- added *BaseResponse.freeze*
- added *werkzeug.contrib.atom*
- the HTTPExceptions accept an argument *description* now which overrides the default description.
- the *MapAdapter* has a default for path info now. If you use *bind_to_environ* you don't have to pass the path later.
- the wsgiref subclass werkzeug uses for the dev server does not use direct sys.stderr logging any more but a logger called "werkzeug".
- implemented *Href*.
- implemented *find_modules*
- refactored request and response objects into base objects, mixins and full featured subclasses that implement all mixins.
- added simple user agent parser
- werkzeug's routing raises *MethodNotAllowed* now if it matches a rule but for a different method.
- many fixes and small improvements

25.1.34 Version 0.1

Released on Dec 9th 2007, codename Wictorinoxger.

- Initial release

25.2 API Changes

0.9

- Soft-deprecated the `BaseRequest.data` and `BaseResponse.data` attributes and introduced new methods to interact with entity data. This will allows in the future to make better APIs to deal with request and response entity bodies. So far there is no deprecation warning but users are strongly encouraged to update.
- The `Headers` and `EnvironHeaders` datastructures are now designed to operate on unicode data. This is a backwards incomaptible change and was necessary for the Python 3 support.
- The `Headers` object no longer supports in-place operations through the old `linked` method. This has been removed without replacement due to changes on the encoding model.

0.6.2

- renamed the attribute *implicit_seqence_conversion* attribute of the request object to *implicit_sequence_conversion*. Because this is a feature that is typically unused and was only in there for the 0.6 series we consider this a bug that does not require backwards compatibility support which would be impossible to properly implement.

0.6

- Old deprecations were removed.
- *cached_property.writeable* was deprecated.
- `BaseResponse.get_wsgi_headers()` replaces the older *BaseResponse.fix_headers* method. The older method stays around for backwards compatibility reasons until 0.7.
- *BaseResponse.header_list* was deprecated. You should not need this function, *get_wsgi_headers* and the *to_list* method on the regular headers should serve as a replacement.
- Deprecated *BaseResponse.iter_encoded*'s charset parameter.
- `LimitedStream` non-silent usage was deprecated.
- the *__repr__* of HTTP exceptions changed. This might break doctests.

0.5

- Werkzeug switched away from wsgiref as library for the builtin webserver.
- The *encoding* parameter for `Templates` is now called *charset*. The older one will work for another two versions but warn with a `DeprecationWarning`.
- The `Client` has cookie support now which is enabled by default.
- `BaseResponse._get_file_stream()` is now passed more parameters to make the function more useful. In 0.6 the old way to invoke the method will no longer work. To support both newer and older Werkzeug versions you can add all arguments to the signature and provide default values for each of them.
- `url_decode()` no longer supports both *&* and *;* as separator. This has to be specified explicitly now.
- The request object is now enforced to be read-only for all attributes. If your code relies on modifications of some values makes sure to create copies of them using the mutable counterparts!
- Some data structures that were only used on request objects are now immutable as well. (`Authorization` / `Accept` and subclasses)
- *CacheControl* was splitted up into `RequestCacheControl` and `ResponseCacheControl`, the former being immutable. The old class will go away in 0.6
- undocumented *werkzeug.test.File* was replaced by `FileWrapper`.
- it's not longer possible to pass dicts inside the *data* dict in `Client`. Use tuples instead.
- It's save to modify the return value of `MultiDict.getlist()` and methods that return lists in the `MultiDict` now. The class creates copies instead of revealing the internal lists. However `MultiDict.setlistdefault` still (and intentionally) returns the internal list for modifications.

0.3

- Werkzeug 0.3 will be the last release with Python 2.3 compatibility.
- The *environ_property* is now read-only by default. This decision was made because the request in general should be considered read-only.

0.2

- The *BaseReporterStream* is now part of the contrib module, the new module is *werkzeug.contrib.reporterstream*. Starting with *0.3*, the old import will not work any longer.
- *RequestRedirect* now uses a 301 status code. Previously a 302 status code was used incorrectly. If you want to continue using this 302 code, use `response = redirect(e.new_url, 302)`.
- *lazy_property* is now called *cached_property*. The alias for the old name will disappear in Werkzeug 0.3.

- *match* can now raise *MethodNotAllowed* if configured for methods and there was no method for that request.
- The *response_body* attribute on the response object is now called *data*. With Werkzeug 0.3 the old name will not work any longer.
- The file-like methods on the response object are deprecated. If you want to use the response object as file like object use the *Response* class or a subclass of *BaseResponse* and mix the new *ResponseStreamMixin* class and use *response.stream*.

If you can't find the information you're looking for, have a look at the index or try to find it using the search function:

- genindex
- search

INDEX

Symbols

A

B

C

D

G

J

L

M

N

O

P

Q

R

S

T

U

V

W