When Django is too bloated
Specialized Web-Applications with Werkzeug

EuroPython 2017 – Rimini, Italy
Gotthard Base Tunnel
Python is amazing for web developers!

- Bottle
- BlueBream
- CherryPy
- CubicWeb
- Grok
- Nagare
- Pyjs
- Pylons
- TACTIC
- Tornado
- TurboGears
- web2py
- Webware
- Zope 2
Why would I want to use less?

- Learn how stuff works
Why would I want to use less?

- Avoid over-engineering
  - Wastes time and resources
  - Makes updates harder
  - It’s a security risk.
Why would I want to use less?

- You want to do something very specific

- Plan, manage and document chemotherapy treatments
- Built with modern web technology
- Used by hospitals in three European countries
Werkzeug = German for “tool”

- Developed by pocoo team @ pocoo.org
  - Flask, Sphinx, Jinja2
- A “WSGI utility”

- Very lightweight
  - No ORM, No templating engine, etc

- The basis of Flask and others
Werkzeug Features Overview

- WSGI
  - WSGI 1.0 compatible, WSGI Helpers
- Wrapping of requests and responses
- HTTP Utilities
  - Header processing, form data parsing, cookies
- Unicode support
- URL routing system
- Testing tools
  - Testclient, Environment builder
- Interactive Debugger in the Browser
A simple Application

```python
from werkzeug.wrappers import Request, Response

def application(environ, start_response):
    request = Request(environ)
    name = request.args.get('name', 'Rimini!')

    response = Response(f'"Buona sera, {name}!"')
    return response(environ, start_response)

if __name__ == '__main__':
    from werkzeug.serving import run_simple
    run_simple('localhost', 4000, application)
```
from werkzeug.wrappers import Request, Response

@Request.application
def application(request):
    name = request.args.get('name', 'Rimini!')
    return Response("Buona sera, %s!" % name)

if __name__ == '__main__':
    from werkzeug.serving import run_simple
    run_simple('localhost', 4000, application)
```python
from werkzeug.routing import Map, Rule

url_map = Map([
    Rule('/', endpoint='EuroPython/overview'),
    Rule('/<int:year>/', endpoint='EuroPython/event'),
    Rule('/<any(mon,tue,wed,thu,fri):day>/', endpoint='EuroPython/schedule'),
    Rule('/greeting/<name>/', endpoint='greeter'),
])

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)]
```
Middlewares

- Separate parts of the Application as wsgi apps
- Combine as needed
```python
from myproject.app import MyDatabaseApp, MyRegularApp
from myproject.auth import AuthenticationMiddleware
from myproject.db import DBConnectionMiddleware

from werkzeug.wsgi import DispatcherMiddleware, SharedDataMiddleware
from werkzeug.exceptions import NotFound

def build_application():
    # build Database aware part
    app = DBConnectionMiddleware(MyDatabaseApp)

    # Use dispatcher to connect three parts of app
    app = DispatcherMiddleware(app, {
        NotFound,
        {  
            "/app": app,
            "/noDBapp": MyRegularApp
        }
    })

    # wrap static file server
    app = SharedDataMiddleware(app, {
        "static": "/var/www/staticfiles"
    })

    app = AuthenticationMiddleware(app)

    return app
```
HTTP Utilities

- Work with HTTP dates
  
  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

- Read and dump cookies

- Parse form data

```python
>>> from cStringIO import StringIO
>>> data = '--foo\r\n\nContent-Disposition: form-data; name=test\r\n\n--foo--
\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!'
```
Using the test client

```python
def test_my_app():
    c = Client(test_app, BaseResponse)
    resp = c.get('/')
    assert 200 == resp.status_code
    assert '{"key": "value"}' == resp.data
```
Using the test client - pytest fixtures

```python
from werkzeug.test import Client
from werkzeug.testapp import test_app
from werkzeug.wrappers import BaseResponse

@pytest.fixture
def client():
    return Client(test_app, BaseResponse)

def test_my_app(client):
    resp = client.get(/)
    assert 200 == resp.status_code
    assert "{"key": "value"}" == resp.data
```
from werkzeug.test import Client
from werkzeug.testapp import test_app
from myproject.auth import AuthenticationMiddleware
from werkzeug.wrappers import BaseResponse

@ pytest.fixture
def client():
    return Client(test_app, BaseResponse)

@ pytest.fixture
def auth_client():
    app = AuthenticationMiddleware(test_app)
    return Client(app, BaseResponse)

def test_my_app(auth_client):
    resp = client.get('/'
    # I'm not logged in =(
    assert 401 == resp.status_code
Interactive debugger in the Browser

```
builtins.KeyError

KeyError: 2018
```

```
Traceback (most recent call last)

    return f(*args[-2] + (request,)(*args[-2:]))

File "/home/niklas/Desktop/EP17/repo/main.py", line 12, in application
    city = europthons[int(request.args["year"])]

[console ready]
>>> europthons
>>> request.args["year"]
'2018'
```
Endless possibilities

- Connect to a database with SQLAlchemy
- Use Jinja2 to render documents
- Use Celery to schedule asynchronous tasks
- Talk to 3rd party APIs with requests
- Make syscalls

- Remote control a robot to perform tasks at home