Klein Documentation

Release 17.2.0

Twisted Matrix Labs

Aug 17, 2017

Contents

Klein is a micro-framework for developing production-ready web services with Python. It's built on widely used and well tested components like Werkzeug and Twisted, and has near-complete test coverage.

Klein is developed by a team of contributors on GitHub. We're also commonly in `#twisted.web` on Freenode.

CHAPTER 1

Introduction to Klein

This is an introduction to Klein, going through from creating a simple web site to something supporting AJAX and more.

Introduction – Getting Started

Klein is a micro-framework for developing production-ready web services with Python, built off Werkzeug and Twisted. The purpose of this introduction is to show you how to install, use, and deploy Klein-based web applications.

This Introduction

This introduction is meant as a general introduction to Klein concepts.

Everything should be as self-contained, but not everything may be runnable (for example, code that shows only a specific function).

Installing

Klein is available on PyPI. Run this to install it:

```
pip install klein
```

Note: Since Twisted is a Klein dependency, you need to have the requirements to install that as well. You will need the Python development headers and a working compiler - installing `python-dev` and `build-essential` on Debian, Mint, or Ubuntu should be all you need.

Hello World

The following example implements a web server that will respond with "Hello, world!" when accessing the root directory.

```
from klein import run, route

@route('/')
def home(request):
    return 'Hello, world!'

run("localhost", 8080)
```

This imports `run` and `route` from the Klein package, and uses them directly. It then starts a Twisted Web server on port 8080, listening on the loopback address.

This works fine for basic applications. However, by creating a Klein instance, then calling the `run` and `route` methods on it, you are able to make your routing not global.

```
from klein import Klein
app = Klein()

@app.route('/')
def home(request):
    return 'Hello, world!'

app.run("localhost", 8080)
```

By not using the global Klein instance, you can have different Klein routers, each having different routes, if your application requires that in the future.

Adding Routes

Add more decorated functions to add more routes to your Klein applications.

```
from klein import Klein
app = Klein()

@app.route('/')
def pg_root(request):
    return 'I am the root page!'

@app.route('/about')
def pg_about(request):
    return 'I am a Klein application!'

app.run("localhost", 8080)
```

Variable Routes

You can also make *variable routes*. This gives your functions extra arguments which match up with the parts of the routes that you have specified. By using this, you can implement pages that change depending on this – for example, by displaying users on a site, or documents in a repository.

```
from klein import Klein
app = Klein()

@app.route('/user/<username>')
def pg_user(request, username):
    return 'Hi %s!' % (username,)

app.run("localhost", 8080)
```

If you start the server and then visit `http://localhost:8080/user/bob`, you should get `Hi bob!` in return.

You can also define what types it should match. The three available types are `string` (default), `int` and `float`.

```
from klein import Klein
app = Klein()

@app.route('/<string:arg>')
def pg_string(request, arg):
    return 'String: %s!' % (arg,)

@app.route('/<float:arg>')
def pg_float(request, arg):
    return 'Float: %s!' % (arg,)

@app.route('/<int:arg>')
def pg_int(request, arg):
    return 'Int: %s!' % (arg,)

app.run("localhost", 8080)
```

If you run this example and visit `http://localhost:8080/somestring`, it will be routed by `pg_string`, `http://localhost:8080/1.0` will be routed by `pg_float` and `http://localhost:8080/1` will be routed by `pg_int`.

Route Order Matters

But remember: order matters! This becomes very important when you are using variable paths. You can have a general, variable path, and then have hard coded paths over the top of it, such as in the following example.

```
from klein import Klein
app = Klein()

@app.route('/user/<username>')
def pg_user(request, username):
    return 'Hi %s!' % (username,)

@app.route('/user/bob')
def pg_user_bob(request):
    return 'Hello there bob!'

app.run("localhost", 8080)
```

The later applying route for bob will overwrite the variable routing in `pg_user`. Any other username will be routed to `pg_user` as normal.

Static Files

To serve static files from a directory, set the `branch` keyword argument on the route you're serving them from to `True`, and return a t.w.static.File with the path you want to serve.

```
from twisted.web.static import File
from klein import Klein
app = Klein()

@app.route('/', branch=True)
def pg_index(request):
    return File('./')

app.run("localhost", 8080)
```

If you run this example and then visit `http://localhost:8080/`, you will get a directory listing.

Streamlined Apps With HTML and JSON

For a typical web application, the first order of business is generating some simple HTML pages that users can interact with and that search engines can easily index.

In such an app, you'll want a consistent frame for all pages, something that puts appropriate things into the `<head>` tag, like a title, references to stylesheets and JavaScript functions, and so on. Then, each page has its own distinct content.

While just a little HTML might have been fine for the 90s, modern web apps quickly - sometimes immediately - outgrow HTML though; soon you'll want some way to get just the data from your backend out via a JSON API, often from a dynamic JavaScript or Python front-end in the browser.

Klein provides for this general pattern with `klein.Plating`.

Let's build a little app that gives us some fake (random) information about places you can go and foods you can get there. You can download the full example `here` in order to run it.

First, we'll create a top-level Plating for the site. This takes a `twisted.web.template` template, defined with the objects from `twisted.web.template.tags`, with one special slot, named `Plating.CONTENT`, in the spot where you want the content of each page to appear. That'll look something like this:

```
from twisted.web.template import tags, slot
from klein import Klein, Plating
app = Klein()

myStyle = Plating(
    tags=tags.html(
        tags.head(tags.title(slot("pageTitle"))),
        tags.body(tags.h1(slot("pageTitle"), Class="titleHeading"),
                  tags.div(slot(Plating.CONTENT)))
    ),
    defaults={"pageTitle": "Places & Foods"}
)
```

Notice that we have defined a `"pageTitle"` slot in the template - individual pages must each provide a value for the title themselves in order to use the `myStyle` frame. Nothing's special about `"pageTitle"`, by the way; you may define whatever slots you want in your page template.

You can also specify a dictionary of default values to fill slots with.

Next, you want to create a route that is plated with that `Plating`, by using the `Plating.routed` decorator. `@myStyle.routed` takes a route from the Klein instance, in this case `app`, and then a template for the content portion (the `Plating.CONTENT` slot) of the page. The decorated function must then return a dictionary of the values to populate the slots in the template with.

Let's start with a really simple page that just has a static template to fill the content slot.

```
@myStyle.routed(
    app.route("/"),
    tags.div(
        tags.h2("Sample Places:"),
        tags.ul([tags.li(tags.a(href=["/places/", place])(place))
                 for place in ["new york", "san francisco", "shanghai"]]),
        tags.h2("Sample Foods:"),
        tags.ul([tags.li(tags.a(href=["/foods/", food])(food))
                 for food in ["hamburgers", "cheeseburgers", "hot dogs"]]),
    ))
def root(request):
    return {}
```

This page generates some links to various sub-pages which we'll get to in a moment. But first, if you load `http://localhost:8080/`, you'll see that the template specified for `root` is inserted at the point in the template for `myStyle` specified the content should go.

Next, we should actually try injecting some data.

```
@myStyle.routed(app.route("/foods/<food>"),
                tags.table(border="2", style="color: blue")(
                    tags.tr(tags.td("Name:"), tags.td(slot("name"))),
                    tags.tr(tags.td("Deliciousness:"),
                            tags.td(slot("rating"), " stars")),
                    tags.tr(tags.td("Carbs:"),
                            tags.td(slot("carbohydrates")))))
def one_food(request, food):
    random = random_from_string(food)
    return {"name": food,
            "pageTitle": "Food: {}".format(food),
            "rating": random.randint(1, 5),
```

Here you can see the `/foods/...` route for showing information about a food. In the content template, we've got slots for `"name"`, `"rating"`, and `"carbohydrates"`, the three primary properties which define a food. The decorated function then returns a dictionary that returns values for each of those slots, as well as a value for `"pageTitle"`.

Each of these slots is only filled with a single item, though. What if you need to put multiple items into the template? The route for `/places/...` can show us:

```
@myStyle.routed(
    app.route("/places/<place>"),
    tags.div(style="color: green")(
        tags.h1("Place: ", slot("name")),
        tags.div(slot("latitude"), "° ", slot("longitude"), "°"),
        tags.div(tags.h2("Foods Found Here:"),
                 tags.ul(tags.li(render="foods:list")(
                     tags.a(href=["/foods/", slot("item")])(slot("item")))))))
def one_place(request, place):
    random = random_from_string(place)
    possible_foods = ["hamburgers", "cheeseburgers", "hot dogs", "pizza",
```

```
                      "", "", "foie gras"]
    random.shuffle(possible_foods)
    return {"name": place,
            "pageTitle": "Place: {}".format(place),
            "latitude": random.uniform(-90, 90),
```

Here you can see the special `<slotname>:list` renderer in use. By specifying the `render=` attribute of a tag (in this case, a `li` tag) to be `foods:list`, we invoke a `twisted.web.template` renderer that repeats the tag it is the renderer for, inserting each element of that list into the special `"item"` slot.

You can view each of these pages in a web browser now, and you can see their contents; we've built a little website that generates random values for these types of data. But we've *also* built a JSON API. If you access, for example, `http://localhost:8080/places/chicago`, you'll see an HTML view, but if you add the query parameter `json=1` (e.g. `http://localhost:8080/places/chicago?json=1`) you will see a JSON result like this:


```
{
    "foods": [
        "pizza",
        "cheeseburgers",
        "hot dogs"
    ],
    "latitude": -32.610538480748815,
    "longitude": -9.38433633489143,
    "name": "chicago",
    "pageTitle": "Place: chicago"
}
```


Any route decorated by `@routed` will similarly give you structured data if you ask for it via `?json=1`, so you can build your JSON API and your HTML frontend at the same time.

Deferreds

Since it's all just Twisted underneath, you can return Deferreds, which then fire with a result.

```
import treq
from klein import Klein
app = Klein()

@app.route('/', branch=True)
def google(request):
    d = treq.get('https://www.google.com' + request.uri)
    d.addCallback(treq.content)
    return d

app.run("localhost", 8080)
```

This example here uses treq (think Requests, but using Twisted) to implement a Google proxy.

Return Anything

Klein tries to do the right thing with what you return. You can return a result (which can be regular text, a Resource, or a Renderable) synchronously (via `return`) or asynchronously (via `Deferred`). Just remember not to give Klein any `unicode`, you have to encode it into `bytes` first.

Onwards

That covers most of the general Klein concepts. The next chapter is about deploying your Klein application using Twisted's `tap` functionality.

Introduction – Using `twistd` to Start Your Application

`twistd` (pronounced "twist-dee") is an application runner for Twisted applications. It takes care of starting your app, setting up loggers, daemonising, and providing a nice interface to start it.

Using the `twistd web` Plugin

Exposing a valid IResource will allow your application to use the pre-existing `twistd web` plugin.

To enable this functionality, just expose the `resource` object of your Klein router:

```
from klein import Klein
app = Klein()

@app.route('/')
def hello(request):
    return "Hello, world!"

resource = app.resource
```

Then run it (in this example, the file above is saved as `twistdPlugin.py`:

```
$ twistd -n web --class=twistdPlugin.resource
```

The full selection of options you can give to `twistd web` can be found in its help page. Here are some relevant entries in it:

```
  -n, --notracebacks      Do not display tracebacks in broken web pages.
                          Displaying tracebacks to users may be security risk!
  -p, --port=             strports description of the port to start the server
                          on.
  -l, --logfile=          Path to web CLF (Combined Log Format) log file.
      --https=            Port to listen on for Secure HTTP.
  -c, --certificate=      SSL certificate to use for HTTPS. [default:
                          server.pem]
  -k, --privkey=          SSL certificate to use for HTTPS. [default:
                          server.pem]
      --class=            Create a Resource subclass with a zero-argument
                          constructor.
```

Using HTTPS via the `twistd web` Plugin

The `twistd web` plugin has inbuilt support for HTTPS, assuming you have TLS support for Twisted.

As an example, we will create some self-signed certs – for the second command, the answers don't really matter, as this is only a demo:

```
$ openssl genrsa > privkey.pem
$ openssl req -new -x509 -key privkey.pem -out cert.pem -days 365
```

We will then run our plugin, specifying a HTTPS port and the relevant certificates:

```
$ twistd -n web --class=twistdPlugin.resource -c cert.pem -k privkey.pem --https=4433
```

This will then start a HTTPS server on port 4433. Visiting `https://localhost:4433` will give you a certificate error – if you add a temporary exception, you will then be given the "Hello, world!" page. Inspecting your browser's URL bar should reveal a little lock – meaning that the connection is encrypted!

Of course, in production, you'd be using a cert signed by a certificate authority – but self-signed certs have their uses.

CHAPTER 2

Klein Examples

These are examples that show how to use different parts of Klein, or use things with Klein.

Example – Serving Static Files

Helpfully you can also return a t.w.resource.IResource such as t.w.static.File. If `branch=True` is passed to `route` the returned `IResource` will also be allowed to handle all children path segments. So `http://localhost:8080/static/img.gif` should return an image and `http://localhost:8080/static/` should return a directory listing.

```
from twisted.web.static import File
from klein import run, route

@route('/static/', branch=True)
def static(request):
    return File("./static")

@route('/')
def home(request):
    return '<img src="/static/img.gif">'

run("localhost", 8080)
```

Example – Using Twisted.Web Templates

You can also make easy use of Twisted's templating system by returning anything that implements IRenderable. For example, returning a t.w.template.Element will make it be rendered, with the result sent as the response body:

```
from twisted.web.template import Element, XMLString, renderer
from klein import run, route
```

```
class HelloElement(Element):
    loader = XMLString((
        '<h1 '
        'xmlns:t="http://twistedmatrix.com/ns/twisted.web.template/0.1"'
        '>Hello, <span t:render="name"></span>!</h1>'))

    def __init__(self, name):
        self._name = name

    @renderer
    def name(self, request, tag):
        return self._name


@route('/hello/<string:name>')
def home(request, name='world'):
    return HelloElement(name)

run("localhost", 8080)
```

Example – Using Deferreds

And of course, this is Twisted. So there is a wealth of APIs that return a Deferred. A Deferred may also be returned from handler functions and their result will be used as the response body.

Here is a simple Google proxy, using treq (think Requests, but using Twisted):

```
import treq
from klein import Klein
app = Klein()

@app.route('/', branch=True)
def google(request):
    d = treq.get(b'https://www.google.com' + request.uri)
    d.addCallback(treq.content)
    return d

app.run("localhost", 8080)
```

Example – Using Async/Await

The previous example which used the Twisted Deferred class can be written to use the new `async` and `await` keywords that are available in Python 3.5+.

Here is the same Google proxy, using treq and `async` and `await`.

```
import treq
from klein import Klein
app = Klein()

@app.route('/', branch=True)
async def google(request):
    response = await treq.get(b'https://www.google.com' + request.uri)
```

```
    content = await treq.content(response)
    return content

app.run("localhost", 8080)
```

Example – Using `twistd`

Another important integration point with Twisted is the twistd application runner. It provides rich logging support, daemonization, reactor selection, profiler integration, and many more useful features.

To provide access to these features (and others like HTTPS) klein provides the `resource` function which returns a valid IResource for your application.

Here is our "Hello, World!" application again in a form that can be launched by `twistd`:

```
from klein import resource, route

@route('/')
def hello(request):
    return "Hello, world!"
```

To run the above application we can save it as `helloworld.py` and use the `twistd web` plugin:

```
PYTHONPATH=. twistd -n web --class=helloworld.resource
```

Example – Handling POST

The `route` decorator supports a `methods` keyword which is the list of HTTP methods as strings. For example, `methods=['POST']` will cause the handler to be invoked when an `POST` request is received. If a handler can support multiple methods the current method can be distinguished with `request.method`.

Here is our `"Hello, world!"` example extended to support setting the name we are saying Hello to via a `POST` request with a `name` argument.

This also demonstrates the use of the redirect method of the request to redirect back to `'/'` after handling the `POST`.

The most specific handler should be defined first. So the `POST` handler must be defined before the handler with no `methods`.

```
from twisted.internet.defer import succeed
from klein import run, route

name='world'

@route('/', methods=['POST'])
def setname(request):
    global name
    name = request.args.get('name', ['world'])[0]
    request.redirect('/')
    return succeed(None)

@route('/')
def hello(request):
    return "Hello, {0}!".format(name)
```

```
run("localhost", 8080)
```

The following curl command can be used to test this behaviour:

```
curl -v -L -d name='bob' http://localhost:8080/
```

Accessing the request content

To read the content of the request use the `read` method of IRequest.content

```
from klein import run, route
import json

@route('/', methods=['POST'])
def do_post(request):
    content = json.loads(request.content.read())
    response = json.dumps(dict(the_data=content), indent=4)
    return response

run("localhost", 8080)
```

The following curl command can be used to test this behaviour:

```
curl -XPOST -v -H 'Content-Type: appliction/json' -d '{"name":"bob"}'  http://
↪localhost:8080/
```

Example – Subroutes

The `routes` decorator lets you set up different routes easily, but it can be cumbersome to write many similar routes. If you need to write similar routes, the `subroute` function can help. `subroute` is a context manager within whose scope all invocations of `route` will have a prefix added.

Here is an example app that has routes for `/branch/lair`, `/branch/crypt`, and `/branch/swamp` all defined in a `with app.subroute()` block.

```
from klein import Klein

app = Klein()

with app.subroute("/branch") as app:
    @app.route("/lair")
    def lair(request):
        return b"These stairs lead to the lair of beasts."

    @app.route("/crypt")
    def crypt(request):
        return b"These stairs lead to an ancient crypt."

    @app.route("/swamp")
    def swamp(request):
        return b"A stair to a swampy wasteland."

app.run("localhost", 8080)
```

Example – Using Non-Global State

For obvious reasons it may be desirable for your application to have some non-global state that is used by your route handlers.

Below we have created a simple `ItemStore` class that has an instance of `Klein` as a class variable `app`. We can now use `@app.route` to decorate the methods of the class.

```
import json

from klein import Klein


class ItemStore(object):
    app = Klein()

    def __init__(self):
        self._items = {}

    @app.route('/')
    def items(self, request):
        request.setHeader('Content-Type', 'application/json')
        return json.dumps(self._items)

    @app.route('/<string:name>', methods=['PUT'])
    def save_item(self, request, name):
        request.setHeader('Content-Type', 'application/json')
        body = json.load(request.content)
        self._items[name] = body
        return json.dumps({'success': True})

    @app.route('/<string:name>', methods=['GET'])
    def get_item(self, request, name):
        request.setHeader('Content-Type', 'application/json')
        return json.dumps(self._items.get(name))


if __name__ == '__main__':
    store = ItemStore()
    store.app.run('localhost', 8080)
```

Example – Handling Errors

It may be desirable to have uniform error-handling code for many routes. We can do this with `Klein.handle_errors`.

Below we have created a class that will translate `NotFound` exceptions into a custom 404 response.

```
from klein import Klein

class NotFound(Exception):
    pass


class ItemStore(object):
    app = Klein()
```

```
    @app.handle_errors(NotFound)
    def notfound(self, request, failure):
        request.setResponseCode(404)
        return 'Not found, I say'

    @app.route('/droid/<string:name>')
    def droid(self, request, name):
        if name in ['R2D2', 'C3P0']:
            raise NotFound()
        return 'Droid found'

    @app.route('/bounty/<string:target>')
    def bounty(self, request, target):
        if target == 'Han Solo':
            return '150,000'
        raise NotFound()


if __name__ == '__main__':
    store = ItemStore()
    store.app.run('localhost', 8080)
```

The following cURL commands (and output) can be used to test this behaviour:

```
curl -L http://localhost:8080/droid/R2D2
Not found, I say

curl -L http://localhost:8080/droid/Janeway
Droid found

curl -L http://localhost:8080/bounty/things
Not found, I say
```

Example - Catch All Routes

A simple way to create a catch-all function, which serves every URL that doesn't match a route, is to use a `path` variable in the route.

```
from klein import Klein

class OnlyOneRoute(object):
    app = Klein()

    @app.route('/api/<path:catchall>')
    def catchAll(self, request, catchall):
        request.redirect('/api')

    @app.route('/api')
    def home(self, request):
        return 'API Home'

    @app.route('/api/v1')
    def v1(self, request):
        return 'Version 1 - Home'
```

```
if __name__ == '__main__':
    oneroute = OnlyOneRoute()
    oneroute.app.run('localhost', 8080)
```

Use cURL to verify that only `/api` and `/api/v1` return content, all other requests are redirected:

```
curl -L http://localhost:8080/api
API Home

curl -L localhost:8080/api/v1
Version 1 - Home

curl -L localhost:8080/api/another
API Home
```

This method can also be used on the root route, in which case it will catch every request which doesn't match a route.

CHAPTER 3

Contributing

If you'd like to help out, here's some material to help you get started!

Contributing to Klein

Klein is an open source project that welcomes contributions of all kinds coming from the community, including:

- code patches,
- documentation improvements,
- bug reports,
- reviews for contributed patches.

Getting started

Here is a list of shell commands that will install the dependencies of Klein, run the test suite with Python 2.7 and the current version of Twisted, compile the documentation, and check for coding style issues with flake8.

```
pip install --user tox
tox -e py27-twcurrent
tox -e docs
tox -e flake8
```

Tox makes a virtualenv, installs Klein's dependencies into the virtualenv, and then runs a set of commands based on the `-e` (environment) argument. This strategy allows one to make and test changes to Klein without needing to change system-level Python packages.

Next steps

Here are some suggestions to make the contributing process easier for everyone:

Code

- Use Twisted's coding standards as a guideline for code changes you make. Some parts of Klein (eg. `klein.resource.ensure_utf8_bytes`) do not adhere to the Twisted style guide, but changing that would break public APIs, which is worse than breaking the style guidelines. Similarly, if you change existing code, following the Twisted style guide is good, but is less important than not breaking public APIs.
- Compatibility across versions is important: here are Twisted's compatibility guidelines, which Klein shares.
- If you're adding a new feature, please add a file with an example and some explanation to the examples directory, then add your example to `/docs/index.rst`.
- Please run `tox -e flake8` to check for style issues in changed code. Flake8 and similar tools expose many small-but-common errors early enough that it's easy to remedy the problem.
- Code changes should have tests: untested code is buggy code. Klein uses Twisted Trial and tox for its tests. The command to run the full test suite is `tox` with no arguments. This will run tests against several versions of Python and Twisted, which can be time-consuming. To run tests against only one or a few versions, pass a `-e` argument with an environment from the envlist in `tox.ini`: for example, `tox -e py35-twcurrent` will run tests with Python 3.5 and the current released version of Twisted. To run only one or a few specific tests in the suite, add a filename or fully-qualified Python path to the end of the test invocation: for example, `tox klein.test.test_app.KleinTestCase.test_foo` will run only the `test_foo()` method of the `KleinTestCase` class in `klein/test/test_app.py`. These two test shortcuts can be combined to give you a quick feedback cycle, but make sure to check on the full test suite from time to time to make sure changes haven't had unexpected side effects.
- Show us your code changes through pull requests sent to Klein's GitHub repo. This is the best way to make your code visible to others and to get feedback about it.
- If your pull request is a work in progress, please put `[WIP]` in its title.
- Add yourself to `AUTHORS`. **Your contribution matters.**

Documentation

Klein uses Epydoc for docstrings in code and uses Sphinx for standalone documentation.

- In documents with headers, use this format and order for headers:

```
========
Header 1
========

Header 2
========

Header 3
--------

Header 4
~~~~~~~~
```

- In prose, please use gender-neutral pronouns or structure sentences such that using pronouns is unecessary.
- It's best to put each sentence on a different line: this makes diffs much easier to read. Sentences that are part of list items need to be indented to be considered part of the same list item.

Reviewing

All code changes added to Klein must be reviewed by at least one other person who is not an author of the code being added. This helps prevent bugs from slipping through the net, gives another source for improvements, and makes sure that the code productively follows guidelines.

Reviewers should read Glyph's mailing list post on reviewing – code and docs don't have to be *perfect*, only *better*.

Releasing Klein

Klein is released on a time-dependent basis, similar to Twisted.

Each version is numbered with the major portion being the last two digits of the year, and the minor portion being the zero-indexed release number. That is, the first release of 2016 would be 16.0, and the second would be 16.1.

Doing a Release

1. Change the version number and commit it.
2. Clear the directory of any other changes using `git clean -f -x -d .`
3. Tag the release using `git tag -s <release> -m "Tag <release> release"`
4. Push up the tag using `git push --tags`.
5. Make a pull request for this changes. Continue when it is merged.
6. Generate the tarball and wheel using `python setup.py sdist bdist_wheel`.
7. Upload the tarball and wheel using `twine upload dist/klein-*`.