yli

Release 0.1.0

Laura F. Dickinson

Nov 17, 2019

CONTENTS

`yli` is a simplistic Python 3.7+ library for writing web apps.

yli operates on the guiding principle of: **Your web server is a function.** This means that your web app can be represented with the following signature:

```
async def app(request: Request) -> Response:
    ...
```

CHAPTER

ONE

INSTALLATION

yli is available on PyPI under `yli`:

```
$ poetry add yli
```

Or for the latest development version:

```
$ poetry add git+https://github.com/Fuyukai/yli.git
```

1.1 Guide

In yli, all web applications are simply represented by a single root function that takes a `Request`, and returns a `Response`. Any complexity such as routing can be represented as these functions calling other functions.

These functions are called `application functions`. You can also have a class be an application function by giving an `async def __call__(self, req: Request) -> Response:`

1.1.1 Your first app

A simple hello world app would look like the following:

```
async def my_app(r: Request) -> Response:
    print("Hello, world!")

async def main():
    webserver = Webserver([BindInfo(7777)], my_app)
    await webserver.run()

trio.run(main)
```

Note: For more information on how the bundled webserver works, see *Serving*.

When running this, you'll get a nice `Hello, world!` message in the console when you GET /. But, on the client side, you'll get:

```
$ http GET http://127.0.0.1:7777/

HTTP/1.1 500 Internal Server Error
content-length: 41
```

(continues on next page)

(continued from previous page)

```
date: Sun, 17 Nov 2019 03:08:35 GMT
server: yli (python/3.8.0)

Response function didn't return anything!
```

Your response function needs to always return a response. You can do this with the `Request.respond()` helper function. Change your request handler to:

```
async def my_app(r: Request) -> Response:
    return r.respond(body="Hello, world!", status_code=200)
```

Now when you do a HTTP reequest, you'll get a nice response:

```
$ http GET http://127.0.0.1:7777/

HTTP/1.1 200 OK
content-length: 13
date: Sun, 17 Nov 2019 03:12:37 GMT
server: yli (python/3.8.0)

Hello, world!
```

1.1.2 Routing

You may note that if you try any other route other than `/`, you still get a plain `Hello, world!` response:

```
$ http GET http://127.0.0.1:7777/abc

HTTP/1.1 200 OK
content-length: 13
date: Sun, 17 Nov 2019 03:13:19 GMT
server: yli (python/3.8.0)

Hello, world!
```

This is because your function doesn't care what is passed in - it's a plain function. However, yli includes some callables that do care what is passed in - such as the `Router`. The router can be used for routing certain paths to functions.

```
from yli.compose import Router

async def hello(request: Request) -> Response:
    return request.respond(body="Hello!")

async def goodbye(request: Request) -> Response:
    return request.respond(body="Goodbye!")

async def main():
    router = Router()
    # you could also chain these calls, router.route(...).route(...)
    router.route("/hello", hello)
    router.route("/goodbye", goodbye)

    await Webserver([BindInfo(7777)], router).run()
```

Now your app will respond the appropriate routes with the right response:

```
$ http GET http://127.0.0.1:7777/hello
HTTP/1.1 200 OK
content-length: 6
date: Sun, 17 Nov 2019 03:17:48 GMT
server: yli (python/3.8.0)

Hello!

$ http GET http://127.0.0.1:7777/goodbye
HTTP/1.1 200 OK
content-length: 8
date: Sun, 17 Nov 2019 03:17:50 GMT
server: yli (python/3.8.0)

Goodbye!

$ http GET http://127.0.0.1:7777/unknown
HTTP/1.1 404 Not Found
content-length: 62
date: Sun, 17 Nov 2019 03:18:42 GMT
server: yli (python/3.8.0)

Couldn't match route (full route: /unknown, working: /unknown)
```

404s

To override 404s, just pass an application function as the `no_route_fun` argument:

```
async def handle_no_route(req: Request) -> Response:
    return req.respond(body="Not found! Go away!", status_code=404)

async def main():
    router = Router(no_route_fun=handle_no_route).route(...)
```

```
$ http GET http://127.0.0.1:7777/unknown
HTTP/1.1 404 Not Found
content-length: 19
date: Sun, 17 Nov 2019 03:21:44 GMT
server: yli (python/3.8.0)

Not found! Go away!
```

1.1.3 Subrouting

Since a router counts as an application function, you can nest them easily.

```
async def xyz(request: Request) -> Response:
    return request.respond(body="123")

async def main():
    rooter = Router()
    nested_router = Router().route("/xyz", xyz)
    rooter.route("/abc", nested_router)
```

Now, requesting on `/abc/xyz` will call your xyz function. Subrouters also have their own 404 handlers, so `/abc/anything_else` will call the subrouter's handler.

```
$ http GET http://127.0.0.1:7777/abc/xyz
HTTP/1.1 200 OK
content-length: 3
date: Sun, 17 Nov 2019 03:25:37 GMT
server: yli (python/3.8.0)

123
```

1.1.4 Method matching

You'll notice in these last examples that your functions completely ignored methods. GET or POST would both call the same function. Often, you don't want that. Enter the `MethodMatcher`, a helper application function which matches up methods to your functions.

```
from yli.compose import MethodMatcher

async def abc_get(q: Request) -> Response:
    return q.respond(body="I've been get!")

async def abc_post(request: Request) -> Response:
    return request.respond(body="I've been posted!")

async def main():
    # you can do method=fn in the constructor...
    matcher = MethodMatcher(get=abc_get, post=abc_post)
    # or chain calls:
    matcher = MethodMatcher().match("get", abc_get).match("post", abc_post)
```

```
$ http GET http://127.0.0.1:7777/
HTTP/1.1 200 OK
content-length: 14
date: Sun, 17 Nov 2019 03:29:37 GMT
server: yli (python/3.8.0)

I've been get!

$ http POST http://127.0.0.1:7777/
HTTP/1.1 200 OK
content-length: 17
date: Sun, 17 Nov 2019 03:29:40 GMT
server: yli (python/3.8.0)

I've been posted!

$ http PUT http://127.0.0.1:7777/
HTTP/1.1 405 Method Not Allowed
content-length: 23
date: Sun, 17 Nov 2019 03:29:46 GMT
server: yli (python/3.8.0)

Method not allowed: PUT
```

Like with `Router`, you can override the default 405 behaviour by passing `invalid_method_fun` to

`MethodMatcher`.

Since `MethodMatcher` is an application fun, you can pass it to helpers like `Router` to only allow specific methods on a route, for example.

1.1.5 Path parameters

Sometimes you might want a parameter in the URL path to be passed to you, for example an ID in a RESTful API. For this, `PathParam` exists.

```
from yli.compose import PathParam

async def print_userid(r: Request) -> Response:
    print(r.path_params["userid"]
    return r.respond(body="OK")

async def main():
    router = Router()
    userid_handler = PathParam("userid", print_userid)  # you could pass a router to␣
↪subroute!
    router.route("/api/v1/user", userid_handler)
    await Webserver(..., router).run()
```

1.1.6 Chaining application functions

It is possible to chain application functions together using `Chain`.

```
from yli.compose import chain, print_response

chained = Chain(my_func).then(print_response)
```

But wait! How can `print_request` be `(Request) -> Response` if `my_func` should return `Response`? I lied a bit - only one application function is allowed within a `Chain`. It is used to add request pre-processors or response post-processors instead. Each function BEFORE the application function should be a `(Request) -> Request`, and each function after it should be a `(Response) -> Response`.

For example, to use `print_request()` to print the result of all responses, you can chain it after your main application function:

```
rooter = Router().route(...)
app = Chain(rooter).then(print_response)
await Webserver(..., app).run()
```

1.1.7 Pre-conditions

Whilst `Chain` can be used to transform requests/responses appropriately, it can't be used to require certain parts of an input. That is where `ShortCircuit` comes in. Functions passed to the `.next()` of `ShortCircuit` can either return a request (optionally, transformed) or return a response to stop the chain early.

```
async def requires_header(r: Request) -> Union[Request, Response]:
    if not "my-secret-header" in r.headers:
        return r.response(body="Not allowed!", code=401)
```

(continues on next page)

(continued from previous page)

```
    return r

app = ShortCircuit().next(requires_headers).last(my_app_router)
```

1.1.8 Writing your own composers

yli has effectively zero magic - basically everything is an application function, so you can compose them relatively freely. To write your own composer, simply have it be an application function that takes another function, and call it (or maybe don't!) and return the appropriate response objects.

1.2 Serving

1.2.1 Built-in

yli includes a trio-powered webserver that can be used to serve your app. `Webserver` takes a list of `BindInfo`, and a single parameter that is your application function.

```
from yli.http import Webserver, BindInfo

async def main():
    webserver = Webserver([BindInfo(7777)], app)
    await webserver.run()

trio.run(main)
```

1.2.2 SSL

SSL support can be enabled by passing a `SSLContext` to BindInfo:

```
ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ctx.load_cert_chain("cert.pem", "key.pem")

s = Webserver([BindInfo(7777), BindInfo(7778, ssl_context=ctx)], app)
await s.run()
```