FastHTML

Explanations > Routes

Routes

Behaviour in FastHTML apps is defined by routes. The syntax is largely the same as the wonderful FastAPI (which is what you should be using instead of this if you're creating a JSON service. FastHTML is mainly for making HTML web apps, not APIs).

> **Unfinished**
>
> We haven't yet written complete documentation of all of FastHTML's routing features – until we add that, the best place to see all the available functionality is to look over the tests

Note that you need to include the types of your parameters, so that `FastHTML` knows what to pass to your function. Here, we're just expecting a string:

```
from fasthtml.common import *
```

```
app = FastHTML()

@app.get('/user/{nm}')
def get_nm(nm:str): return f"Good day to you, {nm}!"
```

Normally you'd save this into a file such as main.py, and then run it in `uvicorn` using:

```
uvicorn main:app
```

However, for testing, we can use Starlette's `TestClient` to try it out:

```
from starlette.testclient import TestClient
```

```
client = TestClient(app)
r = client.get('/user/Jeremy')
r
```

```
<Response [200 OK]>
```

TestClient uses `httpx` behind the scenes, so it returns a `httpx.Response`, which has a `text` attribute with our response body:

```
r.text
```

```
'Good day to you, Jeremy!'
```

In the previous example, the function name (`get_nm`) didn't actually matter – we could have just called it `_`, for instance, since we never actually call it directly. It's just called through HTTP. In fact, we often do call our

functions `_` when using this style of route, since that's one less thing we have to worry about, naming.

An alternative approach to creating a route is to use `app.route` instead, in which case, you make the function name the HTTP method you want. Since this is such a common pattern, you might like to give a shorter name to `app.route` – we normally use `rt`:

```
rt = app.route

@rt('/')
def post(): return "Going postal!"

client.post('/').text
```

```
'Going postal!'
```

Route-specific functionality

FastHTML supports custom decorators for adding specific functionality to routes. This allows you to implement authentication, authorization, middleware, or other custom behaviors for individual routes.

Here's an example of a basic authentication decorator:

```
from functools import wraps

def basic_auth(f):
    @wraps(f)
    async def wrapper(req, *args, **kwargs):
        token = req.headers.get("Authorization")
        if token == 'abc123':
            return await f(req, *args, **kwargs)
        return Response('Not Authorized', status_code=401)
    return wrapper

@app.get("/protected")
@basic_auth
async def protected(req):
    return "Protected Content"

client.get('/protected', headers={'Authorization': 'abc123'}).text
```

```
'Protected Content'
```

The decorator intercepts the request before the route function executes. If the decorator allows the request to proceed, it calls the original route function, passing along the request and any other arguments.

One of the key advantages of this approach is the ability to apply different behaviors to different routes. You can also stack multiple decorators on a single route for combined functionality.

```
def app_beforeware():
    print('App level beforeware')

app = FastHTML(before=Beforeware(app_beforeware))
client = TestClient(app)
```

```
def route_beforeware(f):
    @wraps(f)
    async def decorator(*args, **kwargs):
        print('Route level beforeware')
        return await f(*args, **kwargs)
    return decorator

def second_route_beforeware(f):
    @wraps(f)
    async def decorator(*args, **kwargs):
        print('Second route level beforeware')
        return await f(*args, **kwargs)
    return decorator

@app.get("/users")
@route_beforeware
@second_route_beforeware
async def users():
    return "Users Page"

client.get('/users').text
```

```
App level beforeware
Route level beforeware
Second route level beforeware
'Users Page'
```

This flexiblity allows for granular control over route behaviour, enabling you to tailor each endpoint's functionality as needed. While app-level beforeware remains useful for global operations, decorators provide a powerful tool for route-specific customization.

Combining Routes

Sometimes a FastHTML project can grow so weildy that putting all the routes into `main.py` becomes unweildy. Or, we install a FastHTML- or Starlette-based package that requires us to add routes.

First let's create a `books.py` module, that represents all the user-related views:

```
# books.py
books_app, rt = fast_app()

books = ['A Guide to FastHTML', 'FastHTML Cookbook', 'FastHTML in 24 Hours']

@rt("/", name="list")
def get():
    return Titled("Books", *[P(book) for book in books])
```

Let's mount it in our main module:

```
from books import app as books_app

app, rt = fast_app(routes=[Mount("/books", books_app, name="books")])
```

```
@rt("/")
def get():
    return Titled("Dashboard",
        P(A(href="/books")("Books")),                    (2)
        Hr(),
        P(A(link=uri("books:list"))("Books")),           (3)
    )

serve()
```

(1) We use `starlette.Mount` to add the route to our routes list. We provide the name of `books` to make discovery and management of the links easier. More on that in items 2 and 3 of this annotations list

(2) This example link to the books list view is hand-crafted. Obvious in purpose, it makes changing link patterns in the future harder

(3) This example link uses the named URL route for the books. The advantage of this approach is it makes management of large numbers of link items easier.

Report an issue