Handling handlers

How handlers work in FastHTML

```
from fasthtml.common import *
from collections import namedtuple
from typing import TypedDict
from datetime import datetime
import json,time
```

```
app = FastHTML()
```

The `FastHTML` class is the main application class for FastHTML apps.

```
rt = app.route
```

`app.route` is used to register route handlers. It is a decorator, which means we place it before a function that is used as a handler. Because it's used frequently in most FastHTML applications, we often alias it as `rt`, as we do here.

Basic Route Handling

```
@rt("/hi")
def get(): return 'Hi there'
```

Handler functions can return strings directly. These strings are sent as the response body to the client.

```
cli = Client(app)
```

`Client` is a test client for FastHTML applications. It allows you to simulate requests to your app without running a server.

```
cli.get('/hi').text
```

```
'Hi there'
```

The `get` method on a `Client` instance simulates GET requests to the app. It returns a response object that has a `.text` attribute, which you can use to access the body of the response. It calls `httpx.get` internally – all httpx HTTP verbs are supported.

```
@rt("/hi")
def post(): return 'Postal'
cli.post('/hi').text
```

```
'Postal'
```

Handler functions can be defined for different HTTP methods on the same route. Here, we define a `post` handler for the `/hi` route. The `Client` instance can simulate different HTTP methods, including POST requests.

Request and Response Objects

```
@app.get("/hostie")
def show_host(req): return req.headers['host']
cli.get('/hostie').text
```

```
'testserver'
```

Handler functions can accept a `req` (or `request`) parameter, which represents the incoming request. This object contains information about the request, including headers. In this example, we return the `host` header from the request. The test client uses 'testserver' as the default host.

In this example, we use `@app.get("/hostie")` instead of `@rt("/hostie")`. The `@app.get()` decorator explicitly specifies the HTTP method (GET) for the route, while `@rt()` by default handles both GET and POST requests.

```
@rt
def yoyo(): return 'a yoyo'
cli.post('/yoyo').text
```

```
'a yoyo'
```

If the `@rt` decorator is used without arguments, it uses the function name as the route path. Here, the `yoyo` function becomes the handler for the `/yoyo` route. This handler responds to GET and POST methods, since a specific method wasn't provided.

```
@rt
def ft1(): return Html(Div('Text.'))
print(cli.get('/ft1').text)
```

```
<html>
  <div>Text.</div>
</html>
```

Handler functions can return `FT` objects, which are automatically converted to HTML strings. The `FT` class can take other `FT` components as arguments, such as `Div`. This allows for easy composition of HTML elements in your responses.

```
@app.get
def autopost(): return Html(Div('Text.', hx_post=yoyo.rt()))
print(cli.get('/autopost').text)
```

```
<html>
  <div hx-post="/yoyo">Text.</div>
</html>
```

The `rt` decorator modifies the `yoyo` function by adding an `rt()` method. This method returns the route path associated with the handler. It's a convenient way to reference the route of a handler function dynamically.

In the example, `yoyo.rt()` is used as the value for `hx_post`. This means when the div is clicked, it will trigger an HTMX POST request to the route of the `yoyo` handler. This approach allows for flexible, DRY code by avoiding hardcoded route strings and automatically updating if the route changes.

This pattern is particularly useful in larger applications where routes might change, or when building reusable components that need to reference their own routes dynamically.

```python
@app.get
def autoget(): return Html(Body(Div('Text.', cls='px-2', hx_post=show_host.rt(a='b'))))
print(cli.get('/autoget').text)
```

```
<html>
  <body>
    <div hx-post="/hostie?a=b" class="px-2">Text.</div>
  </body>
</html>
```

The `rt()` method of handler functions can also accept parameters. When called with parameters, it returns the route path with a query string appended. In this example, `show_host.rt(a='b')` generates the path `/hostie?a=b`.

The `Body` component is used here to demonstrate nesting of FT components. `Div` is nested inside `Body`, showcasing how you can create more complex HTML structures.

The `cls` parameter is used to add a CSS class to the `Div`. This translates to the `class` attribute in the rendered HTML. (`class` can't be used as a parameter name directly in Python since it's a reserved word.)

```python
@rt('/ft2')
def get(): return Title('Foo'),H1('bar')
print(cli.get('/ft2').text)
```

```
<!doctype html>

<html>
  <head>
    <title>Foo</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, viewport-
fit=cover">
    <script src="https://unpkg.com/htmx.org@next/dist/htmx.min.js"></script>
    <script src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-
js@main/fasthtml.js"></script>
    <script src="https://cdn.jsdelivr.net/gh/answerdotai/surreal@main/surreal.js">
</script>
    <script src="https://cdn.jsdelivr.net/gh/gnat/css-scope-inline@main/script.js">
</script>
  </head>
  <body>
    <h1>bar</h1>
  </body>
</html>
```

Handler functions can return multiple `FT` objects as a tuple. The first item is treated as the `Title`, and the rest are added to the `Body`. When the request is not an HTMX request, FastHTML automatically adds necessary HTML boilerplate, including default `head` content with required scripts.

When using `app.route` (or `rt`), if the function name matches an HTTP verb (e.g., `get`, `post`, `put`, `delete`), that HTTP method is automatically used for the route. In this case, a path must be explicitly provided as an argument to the decorator.

```
hxhdr = {'headers':{'hx-request':"1"}}
print(cli.get('/ft2', **hxhdr).text)
```

```
<title>Foo</title>

<h1>bar</h1>
```

For HTMX requests (indicated by the `hx-request` header), FastHTML returns only the specified components without the full HTML structure. This allows for efficient partial page updates in HTMX applications.

```
@rt('/ft3')
def get(): return H1('bar')
print(cli.get('/ft3', **hxhdr).text)
```

```
<h1>bar</h1>
```

When a handler function returns a single `FT` object for an HTMX request, it's rendered as a single HTML partial.

```
@rt('/ft4')
def get(): return Html(Head(Title('hi')), Body(P('there')))

print(cli.get('/ft4').text)
```

```
<html>
  <head>
    <title>hi</title>
  </head>
  <body>
    <p>there</p>
  </body>
</html>
```

Handler functions can return a complete `Html` structure, including `Head` and `Body` components. When a full HTML structure is returned, FastHTML doesn't add any additional boilerplate. This gives you full control over the HTML output when needed.

```
@rt
def index(): return "welcome!"
print(cli.get('/').text)
```

```
welcome!
```

The `index` function is a special handler in FastHTML. When defined without arguments to the `@rt` decorator, it automatically becomes the handler for the root path (`'/'`). This is a convenient way to define the main page or entry point of your application.

Path and Query Parameters

```
@rt('/user/{nm}', name='gday')
def get(nm:str=''): return f"Good day to you, {nm}!"
cli.get('/user/Alexis').text
```

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

Handler functions can use path parameters, defined using curly braces in the route – this is implemented by Starlette directly, so all Starlette path parameters can be used. These parameters are passed as arguments to the function.

The `name` parameter in the decorator allows you to give the route a name, which can be used for URL generation.

In this example, `{nm}` in the route becomes the `nm` parameter in the function. The function uses this parameter to create a personalized greeting.

```
@app.get
def autolink(): return Html(Div('Text.', link=uri('gday', nm='Alexis')))
print(cli.get('/autolink').text)
```

```
<html>
  <div href="/user/Alexis">Text.</div>
</html>
```

The `uri` function is used to generate URLs for named routes. It takes the route name as its first argument, followed by any path or query parameters needed for that route.

In this example, `uri('gday', nm='Alexis')` generates the URL for the route named 'gday' (which we defined earlier as '/user/{nm}'), with 'Alexis' as the value for the 'nm' parameter.

The `link` parameter in FT components sets the `href` attribute of the rendered HTML element. By using `uri()`, we can dynamically generate correct URLs even if the underlying route structure changes.

This approach promotes maintainable code by centralizing route definitions and avoiding hardcoded URLs throughout the application.

```
@rt('/link')
def get(req): return f"{req.url_for('gday', nm='Alexis')}; {req.url_for('show_host')}"

cli.get('/link').text
```

```
'http://testserver/user/Alexis; http://testserver/hostie'
```

The `url_for` method of the request object can be used to generate URLs for named routes. It takes the route name as its first argument, followed by any path parameters needed for that route.

In this example, `req.url_for('gday', nm='Alexis')` generates the full URL for the route named 'gday', including the scheme and host. Similarly, `req.url_for('show_host')` generates the URL for the 'show_host' route.

This method is particularly useful when you need to generate absolute URLs, such as for email links or API responses. It ensures that the correct host and scheme are included, even if the application is accessed through different domains or protocols.

```
app.url_path_for('gday', nm='Jeremy')
```

```
'/user/Jeremy'
```

The `url_path_for` method of the application can be used to generate URL paths for named routes. Unlike `url_for`, it returns only the path component of the URL, without the scheme or host.

In this example, `app.url_path_for('gday', nm='Jeremy')` generates the path '/user/Jeremy' for the route named 'gday'.

This method is useful when you need relative URLs or just the path component, such as for internal links or when constructing URLs in a host-agnostic manner.

```
@rt('/oops')
def get(nope): return nope
r = cli.get('/oops?nope=1')
print(r)
r.text
```

```
<Response [200 OK]>
/Users/jhoward/Documents/GitHub/fasthtml/fasthtml/core.py:175: UserWarning: `nope
has no type annotation and is not a recognised special name, so is ignored.
  if arg!='resp': warn(f"`{arg} has no type annotation and is not a recognised
special name, so is ignored.")
''
```

Handler functions can include parameters, but they must be type-annotated or have special names (like `req`) to be recognized. In this example, the `nope` parameter is not annotated, so it's ignored, resulting in a warning.

When a parameter is ignored, it doesn't receive the value from the query string. This can lead to unexpected behavior, as the function attempts to return `nope`, which is undefined.

The `cli.get('/oops?nope=1')` call succeeds with a 200 OK status because the handler doesn't raise an exception, but it returns an empty response, rather than the intended value.

To fix this, you should either add a type annotation to the parameter (e.g., `def get(nope: str):`) or use a recognized special name like `req`.

```
@rt('/html/{idx}')
def get(idx:int): return Body(H4(f'Next is {idx+1}.'))
print(cli.get('/html/1', **hxhdr).text)
```

```
 <body>
   <h4>Next is 2.</h4>
 </body>
```

Path parameters can be type-annotated, and FastHTML will automatically convert them to the specified type if possible. In this example, `idx` is annotated as `int`, so it's converted from the string in the URL to an integer.

```
reg_re_param("imgext", "ico|gif|jpg|jpeg|webm")

@rt(r'/static/{path:path}{fn}.{ext:imgext}')
def get(fn:str, path:str, ext:str): return f"Getting {fn}.{ext} from /{path}"

print(cli.get('/static/foo/jph.ico').text)
```

```
Getting jph.ico from /foo/
```

The `reg_re_param` function is used to register custom path parameter types using regular expressions. Here, we define a new path parameter type called "imgext" that matches common image file extensions.

Handler functions can use complex path patterns with multiple parameters and custom types. In this example, the route pattern `r'/static/{path:path}{fn}.{ext:imgext}'` uses three path parameters:

1. `path`: A Starlette built-in type that matches any path segments
2. `fn`: The filename without extension
3. `ext`: Our custom "imgext" type that matches specific image extensions

```
ModelName = str_enum('ModelName', "alexnet", "resnet", "lenet")

@rt("/models/{nm}")
def get(nm:ModelName): return nm

print(cli.get('/models/alexnet').text)
```

```
alexnet
```

We define `ModelName` as an enum with three possible values: "alexnet", "resnet", and "lenet". Handler functions can use these enum types as parameter annotations. In this example, the `nm` parameter is annotated with `ModelName`, which ensures that only valid model names are accepted.

When a request is made with a valid model name, the handler function returns that name. This pattern is useful for creating type-safe APIs with a predefined set of valid values.

```
@rt("/files/{path}")
async def get(path: Path): return path.with_suffix('.txt')
print(cli.get('/files/foo').text)
```

```
foo.txt
```

Handler functions can use `Path` objects as parameter types. The `Path` type is from Python's standard library `pathlib` module, which provides an object-oriented interface for working with file paths. In this example, the `path` parameter is annotated with `Path`, so FastHTML automatically converts the string from the URL to a `Path` object.

This approach is particularly useful when working with file-related routes, as it provides a convenient and platform-independent way to handle file paths.

```
fake_db = [{"name": "Foo"}, {"name": "Bar"}]

@rt("/items/")
def get(idx:int|None = 0): return fake_db[idx]
print(cli.get('/items/?idx=1').text)
```

```
{"name":"Bar"}
```

Handler functions can use query parameters, which are automatically parsed from the URL. In this example, `idx` is a query parameter with a default value of 0. It's annotated as `int|None`, allowing it to be either an integer or None.

The function uses this parameter to index into a fake database (`fake_db`). When a request is made with a valid `idx` query parameter, the handler returns the corresponding item from the database.

```
print(cli.get('/items/').text)
```

```
{"name":"Foo"}
```

When no `idx` query parameter is provided, the handler function uses the default value of 0. This results in returning the first item from the `fake_db` list, which is `{"name":"Foo"}`.

This behavior demonstrates how default values for query parameters work in FastHTML. They allow the API to have a sensible default behavior when optional parameters are not provided.

```
print(cli.get('/items/?idx=g'))
```

```
<Response [404 Not Found]>
```

When an invalid value is provided for a typed query parameter, FastHTML returns a 404 Not Found response. In this example, 'g' is not a valid integer for the `idx` parameter, so the request fails with a 404 status.

This behavior ensures type safety and prevents invalid inputs from reaching the handler function.

```
@app.get("/booly/")
def _(coming:bool=True): return 'Coming' if coming else 'Not coming'
print(cli.get('/booly/?coming=true').text)
print(cli.get('/booly/?coming=no').text)
```

```
Coming
Not coming
```

Handler functions can use boolean query parameters. In this example, `coming` is a boolean parameter with a default value of `True`. FastHTML automatically converts string values like 'true', 'false', '1', '0', 'on', 'off', 'yes', and 'no' to their corresponding boolean values.

The underscore `_` is used as the function name in this example to indicate that the function's name is not important or won't be referenced elsewhere. This is a common Python convention for throwaway or unused variables, and it works here because FastHTML uses the route decorator parameter, when provided, to determine the URL path, not the function name. By default, both `get` and `post` methods can be used in routes that don't specify an http method (by either using `app.get`, `def get`, or the `methods` parameter to `app.route`).

```
@app.get("/datie/")
def _(d:parsed_date): return d
date_str = "17th of May, 2024, 2p"
print(cli.get(f'/datie/?d={date_str}').text)
```

```
2024-05-17 14:00:00
```

Handler functions can use `date` objects as parameter types. FastHTML uses `dateutil.parser` library to automatically parse a wide variety of date string formats into `date` objects.

```
@app.get("/ua")
async def _(user_agent:str): return user_agent
print(cli.get('/ua', headers={'User-Agent':'FastHTML'}).text)
```

```
FastHTML
```

Handler functions can access HTTP headers by using parameter names that match the header names. In this example, `user_agent` is used as a parameter name, which automatically captures the value of the 'User-Agent' header from the request.

The `Client` instance allows setting custom headers for test requests. Here, we set the 'User-Agent' header to 'FastHTML' in the test request.

```
@app.get("/hxtest")
def _(htmx): return htmx.request
print(cli.get('/hxtest', headers={'HX-Request':'1'}).text)

@app.get("/hxtest2")
def _(foo:HtmxHeaders, req): return foo.request
print(cli.get('/hxtest2', headers={'HX-Request':'1'}).text)
```

```
1
1
```

Handler functions can access HTMX-specific headers using either the special `htmx` parameter name, or a parameter annotated with `HtmxHeaders`. Both approaches provide access to HTMX-related information.

In these examples, the `htmx.request` attribute returns the value of the 'HX-Request' header.

```
app.chk = 'foo'
@app.get("/app")
def _(app): return app.chk
print(cli.get('/app').text)
```

```
foo
```

Handler functions can access the `FastHTML` application instance using the special `app` parameter name. This allows handlers to access application-level attributes and methods.

In this example, we set a custom attribute `chk` on the application instance. The handler function then uses the `app` parameter to access this attribute and return its value.

```
@app.get("/app2")
def _(foo:FastHTML): return foo.chk,HttpHeader("mykey", "myval")
```

```
r = cli.get('/app2', **hxhdr)
print(r.text)
print(r.headers)
```

```
foo

Headers({'mykey': 'myval', 'content-length': '4', 'content-type': 'text/html;
charset=utf-8'})
```

Handler functions can access the `FastHTML` application instance using a parameter annotated with `FastHTML`. This allows handlers to access application-level attributes and methods, just like using the special `app` parameter name.

Handlers can return tuples containing both content and `HttpHeader` objects. `HttpHeader` allows setting custom HTTP headers in the response.

In this example:

- We define a handler that returns both the `chk` attribute from the application and a custom header.
- The `HttpHeader("mykey", "myval")` sets a custom header in the response.
- We use the test client to make a request and examine both the response text and headers.
- The response includes the custom header "mykey" along with standard headers like content-length and content-type.

```
@app.get("/app3")
def _(foo:FastHTML): return HtmxResponseHeaders(location="http://example.org")
r = cli.get('/app3')
print(r.headers)
```

```
Headers({'hx-location': 'http://example.org', 'content-length': '0', 'content-type':
'text/html; charset=utf-8'})
```

Handler functions can return `HtmxResponseHeaders` objects to set HTMX-specific response headers. This is useful for HTMX-specific behaviors like client-side redirects.

In this example we define a handler that returns an `HtmxResponseHeaders` object with a `location` parameter, which sets the `HX-Location` header in the response. HTMX uses this for client-side redirects.

```
@app.get("/app4")
def _(foo:FastHTML): return Redirect("http://example.org")
cli.get('/app4', follow_redirects=False)
```

```
<Response [303 See Other]>
```

Handler functions can return `Redirect` objects to perform HTTP redirects. This is useful for redirecting users to different pages or external URLs.

In this example:

- We define a handler that returns a `Redirect` object with the URL "http://example.org".
- The `cli.get('/app4', follow_redirects=False)` call simulates a GET request to the '/app4' route without following redirects.
- The response has a 303 See Other status code, indicating a redirect.

The `follow_redirects=False` parameter is used to prevent the test client from automatically following the redirect, allowing us to inspect the redirect response itself.

```
Redirect.__response__
```

```
<function fasthtml.core.Redirect.__response__(self, req)>
```

The `Redirect` class in FastHTML implements a `__response__` method, which is a special method recognized by the framework. When a handler returns a `Redirect` object, FastHTML internally calls this `__response__` method to replace the original response.

The `__response__` method takes a `req` parameter, which represents the incoming request. This allows the method to access request information if needed when constructing the redirect response.

```
@rt
def meta(): 
    return ((Title('hi'),H1('hi')),
        (Meta(property='image'), Meta(property='site_name')))

print(cli.post('/meta').text)
```

```
 <!doctype html>

 <html>
   <head>
     <title>hi</title>
     <meta property="image">
     <meta property="site_name">
     <meta charset="utf-8">
     <meta name="viewport" content="width=device-width, initial-scale=1, viewport-
fit=cover">
     <script src="https://unpkg.com/htmx.org@next/dist/htmx.min.js"></script>
     <script src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-
js@main/fasthtml.js"></script>
     <script src="https://cdn.jsdelivr.net/gh/answerdotai/surreal@main/surreal.js">
</script>
     <script src="https://cdn.jsdelivr.net/gh/gnat/css-scope-inline@main/script.js">
</script>
   </head>
   <body>
     <h1>hi</h1>
   </body>
 </html>
```

FastHTML automatically identifies elements typically placed in the `<head>` (like `Title` and `Meta`) and positions them accordingly, while other elements go in the `<body>`.

In this example: - `(Title('hi'), H1('hi'))` defines the title and main heading. The title is placed in the head, and the H1 in the body. - `(Meta(property='image'), Meta(property='site_name'))` defines two meta tags, which are both placed in the head.

Form Data and JSON Handling

```
@app.post('/profile/me')
def profile_update(username: str): return username

print(cli.post('/profile/me', data={'username' : 'Alexis'}).text)
r = cli.post('/profile/me', data={})
print(r.text)
r
```

```
Alexis
Missing required field: username
<Response [400 Bad Request]>
```

Handler functions can accept form data parameters, without needing to manually extract it from the request. In this example, `username` is expected to be sent as form data.

If required form data is missing, FastHTML automatically returns a 400 Bad Request response with an error message.

The `data` parameter in the `cli.post()` method simulates sending form data in the request.

```
@app.post('/pet/dog')
def pet_dog(dogname: str = None): return dogname or 'unknown name'
print(cli.post('/pet/dog', data={}).text)
```

```
unknown name
```

Handlers can have optional form data parameters with default values. In this example, `dogname` is an optional parameter with a default value of `None`.

Here, if the form data doesn't include the `dogname` field, the function uses the default value. The function returns either the provided `dogname` or 'unknown name' if `dogname` is `None`.

```
@dataclass
class Bodie: a:int;b:str

@rt("/bodie/{nm}")
def post(nm:str, data:Bodie):
    res = asdict(data)
    res['nm'] = nm
    return res

print(cli.post('/bodie/me', data=dict(a=1, b='foo', nm='me')).text)
```

```
{"a":1,"b":"foo","nm":"me"}
```

You can use dataclasses to define structured form data. In this example, `Bodie` is a dataclass with `a` (int) and `b` (str) fields.

FastHTML automatically converts the incoming form data to a `Bodie` instance where attribute names match parameter names. Other form data elements are matched with parameters with the same names (in this case, `nm`).

Handler functions can return dictionaries, which FastHTML automatically JSON-encodes.

```
@app.post("/bodied/")
def bodied(data:dict): return data

d = dict(a=1, b='foo')
print(cli.post('/bodied/', data=d).text)
```

```
{"a":"1","b":"foo"}
```

`dict` parameters capture all form data as a dictionary. In this example, the `data` parameter is annotated with `dict`, so FastHTML automatically converts all incoming form data into a dictionary.

Note that when form data is converted to a dictionary, all values become strings, even if they were originally numbers. This is why the 'a' key in the response has a string value "1" instead of the integer 1.

```
nt = namedtuple('Bodient', ['a','b'])

@app.post("/bodient/")
def bodient(data:nt): return asdict(data)
print(cli.post('/bodient/', data=d).text)
```

```
{"a":"1","b":"foo"}
```

Handler functions can use named tuples to define structured form data. In this example, `Bodient` is a named tuple with `a` and `b` fields.

FastHTML automatically converts the incoming form data to a `Bodient` instance where field names match parameter names. As with the previous example, all form data values are converted to strings in the process.

```
class BodieTD(TypedDict): a:int;b:str='foo'

@app.post("/bodietd/")
def bodient(data:BodieTD): return data
print(cli.post('/bodietd/', data=d).text)
```

```
{"a":1,"b":"foo"}
```

You can use `TypedDict` to define structured form data with type hints. In this example, `BodieTD` is a `TypedDict` with `a` (int) and `b` (str) fields, where `b` has a default value of 'foo'.

FastHTML automatically converts the incoming form data to a `BodieTD` instance where keys match the defined fields. Unlike with regular dictionaries or named tuples, FastHTML respects the type hints in `TypedDict`, converting values to the specified types when possible (e.g., converting '1' to the integer 1 for the 'a' field).

```
class Bodie2:
    a:int|None; b:str
    def __init__(self, a, b='foo'): store_attr()

@app.post("/bodie2/")
def bodie(d:Bodie2): return f"a: {d.a}; b: {d.b}"
print(cli.post('/bodie2/', data={'a':1}).text)
```

```
a: 1; b: foo
```

Custom classes can be used to define structured form data. Here, `Bodie2` is a custom class with `a` (int|None) and `b` (str) attributes, where `b` has a default value of 'foo'. The `store_attr()` function (from fastcore) automatically assigns constructor parameters to instance attributes.

FastHTML automatically converts the incoming form data to a `Bodie2` instance, matching form fields to constructor parameters. It respects type hints and default values.

```
@app.post("/b")
def index(it: Bodie): return Titled("It worked!", P(f"{it.a}, {it.b}"))

s = json.dumps({"b": "Lorem", "a": 15})
print(cli.post('/b', headers={"Content-Type": "application/json", 'hx-request':"1"}, d
```

```
<title>It worked!</title>

<main class="container">
  <h1>It worked!</h1>
  <p>15, Lorem</p>
</main>
```

Handler functions can accept JSON data as input, which is automatically parsed into the specified type. In this example, `it` is of type `Bodie`, and FastHTML converts the incoming JSON data to a `Bodie` instance.

The `Titled` component is used to create a page with a title and main content. It automatically generates an `<h1>` with the provided title, wraps the content in a `<main>` tag with a "container" class, and adds a `title` to the head.

When making a request with JSON data: - Set the "Content-Type" header to "application/json" - Provide the JSON data as a string in the `data` parameter of the request

Cookies, Sessions, File Uploads, and more

```
@rt("/setcookie")
def get(): return cookie('now', datetime.now())

@rt("/getcookie")
def get(now:parsed_date): return f'Cookie was set at time {now.time()}'

print(cli.get('/setcookie').text)
time.sleep(0.01)
cli.get('/getcookie').text
```

```
'Cookie was set at time 16:53:08.485345'
```

Handler functions can set and retrieve cookies. In this example:

- The `/setcookie` route sets a cookie named 'now' with the current datetime.
- The `/getcookie` route retrieves the 'now' cookie and returns its value.

The `cookie()` function is used to create a cookie response. FastHTML automatically converts the datetime object to a string when setting the cookie, and parses it back to a date object when retrieving it.

```
cookie('now', datetime.now())
```

```
HttpHeader(k='set-cookie', v='now="2024-09-07 16:54:05.275757"; Path=/;
SameSite=lax')
```

The `cookie()` function returns an `HttpHeader` object with the 'set-cookie' key. You can return it in a tuple along with `FT` elements, along with anything else FastHTML supports in responses.

```
app = FastHTML(secret_key='soopersecret')
cli = Client(app)
rt = app.route
```

```
@rt("/setsess")
def get(sess, foo:str=''):
    now = datetime.now()
    sess['auth'] = str(now)
    return f'Set to {now}'

@rt("/getsess")
def get(sess): return f'Session time: {sess["auth"]}'

print(cli.get('/setsess').text)
time.sleep(0.01)

cli.get('/getsess').text
```

```
Set to 2024-09-07 16:56:20.445950
'Session time: 2024-09-07 16:56:20.445950'
```

Sessions store and retrieve data across requests. To use sessions, you should to initialize the FastHTML application with a `secret_key`. This is used to cryptographically sign the cookie used by the session.

The `sess` parameter in handler functions provides access to the session data. You can set and get session variables using dictionary-style access.

```
@rt("/upload")
async def post(uf:UploadFile): return (await uf.read()).decode()

with open('../../CHANGELOG.md', 'rb') as f:
    print(cli.post('/upload', files={'uf':f}, data={'msg':'Hello'}).text[:15])
```

```
# Release notes
```

Handler functions can accept file uploads using Starlette's `UploadFile` type. In this example:

- The `/upload` route accepts a file upload named `uf`.
- The `UploadFile` object provides an asynchronous `read()` method to access the file contents.
- We use `await` to read the file content asynchronously and decode it to a string.

We added `async` to the handler function because it uses `await` to read the file content asynchronously. In Python, any function that uses `await` must be declared as `async`. This allows the function to be run asynchronously, potentially improving performance by not blocking other operations while waiting for the file to be read.

```
app.static_route('.md', static_path='../..')
print(cli.get('/README.md').text[:10])
```

```
# FastHTML
```

The `static_route` method of the FastHTML application allows serving static files with specified extensions from a given directory. In this example:

- `.md` files are served from the `../..` directory (two levels up from the current directory).
- Accessing `/README.md` returns the contents of the README.md file from that directory.

```
help(app.static_route_exts)
```

```
Help on method static_route_exts in module fasthtml.core:

static_route_exts(prefix='/', static_path='.', exts='static') method of
fasthtml.core.FastHTML instance
    Add a static route at URL path `prefix` with files from `static_path` and `exts`
defined by `reg_re_param()`
```

```
app.static_route_exts()
print(cli.get('/README.txt').text[:50])
```

```
These are the source notebooks for FastHTML.
```

The `static_route_exts` method of the FastHTML application allows serving static files with specified extensions from a given directory. By default:

- It serves files from the current directory ('.').
- It uses the 'static' regex, which includes common static file extensions like 'ico', 'gif', 'jpg', 'css', 'js', etc.
- The URL prefix is set to '/'.

The 'static' regex is defined by FastHTML using this code:

```
reg_re_param("static", "ico|gif|jpg|jpeg|webm|css|js|woff|png|svg|mp4|webp|ttf|otf|eot
```

```
@rt("/form-submit/{list_id}")
def options(list_id: str):
    headers = {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Methods': 'POST',
        'Access-Control-Allow-Headers': '*',
    }
    return Response(status_code=200, headers=headers)

print(cli.options('/form-submit/2').headers)
```

```
Headers({'access-control-allow-origin': '*', 'access-control-allow-methods': 'POST',
'access-control-allow-headers': '*', 'content-length': '0', 'set-cookie':
'session_=eyJhdXRoIjogIjIwMjQtMDktMDcgMTY6NTY6MjAuNDQ1OTUwIiwgIm5hbWUiOiAiMiJ9.Ztv60
A.N0b4z5rNg6GT3xCFc8X4DqwcNjQ; path=/; Max-Age=31536000; httponly; samesite=lax'})
```

FastHTML handlers can handle OPTIONS requests and set custom headers. In this example:

- The `/form-submit/{list_id}` route handles OPTIONS requests.
- Custom headers are set to allow cross-origin requests (CORS).
- The function returns a Starlette `Response` object with a 200 status code and the custom headers.

You can return any Starlette Response type from a handler function, giving you full control over the response when needed.

```
def _not_found(req, exc): return Div('nope')

app = FastHTML(exception_handlers={404:_not_found})
cli = Client(app)
rt = app.route

r = cli.get('/')
print(r.text)
```

```
 <!doctype html>

 <html>
   <head>
     <title>FastHTML page</title>
     <meta charset="utf-8">
     <meta name="viewport" content="width=device-width, initial-scale=1, viewport-
fit=cover">
     <script src="https://unpkg.com/htmx.org@next/dist/htmx.min.js"></script>
     <script src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-
js@main/fasthtml.js"></script>
     <script src="https://cdn.jsdelivr.net/gh/answerdotai/surreal@main/surreal.js">
</script>
     <script src="https://cdn.jsdelivr.net/gh/gnat/css-scope-inline@main/script.js">
</script>
   </head>
   <body>
     <div>nope</div>
   </body>
 </html>
```

FastHTML allows you to define custom exception handlers. In this example we defined a custom 404 (Not Found) handler function `_not_found`, which returns a `Div` component with the text 'nope'.

FastHTML includes the full HTML structure, since we returned an HTML partial.

Report an issue