 Tutorials > FastHTML By Example

FastHTML By Example

An introduction to FastHTML from the ground up, with four complete examples

This tutorial provides an alternate introduction to FastHTML by building out example applications. We also illustrate how to use FastHTML foundations to create custom web apps. Finally, this document serves as minimal context for a LLM to turn it into a FastHTML assistant.

Let's get started.

FastHTML Basics [↗](#)

FastHTML is *just Python*. You can install it with `pip install python-fasthtml`. Extensions/components built for it can likewise be distributed via PyPI or as simple Python files.

The core usage of FastHTML is to define routes, and then to define what to do at each route. This is similar to the [FastAPI](#) web framework (in fact we implemented much of the functionality to match the FastAPI usage examples), but where FastAPI focuses on returning JSON data to build APIs, FastHTML focuses on returning HTML data.

Here's a simple FastHTML app that returns a "Hello, World" message:

```
from fasthtml.common import FastHTML, serve

app = FastHTML()

@app.get("/")
def home():
    return "<h1>Hello, World</h1>"

serve()
```

To run this app, place it in a file, say `app.py`, and then run it with `python app.py`.

```
INFO:      Will watch for changes in these directories: ['/home/jonathan/fasthtml-
example']
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [871942] using WatchFiles
INFO:      Started server process [871945]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

If you navigate to <http://127.0.0.1:8000> in a browser, you'll see your "Hello, World". If you edit the `app.py` file and save it, the server will reload and you'll see the updated message when you refresh the page in your

browser.

Constructing HTML

Notice we wrote some HTML in the previous example. We don't want to do that! Some web frameworks require that you learn HTML, CSS, JavaScript AND some templating language AND python. We want to do as much as possible with just one language. Fortunately, the Python module [fastcore.xml](#) has all we need for constructing HTML from Python, and FastHTML includes all the tags you need to get started. For example:

```
from fasthtml.common import *
page = Html(
    Head(Title('Some page')),
    Body(Div('Some text, ', A('A link', href='https://example.com'), Img(src='https://
print(to_xml(page))
```

```
<!doctype html></!doctype>

<html>
  <head>
    <title>Some page</title>
  </head>
  <body>
    <div class="myclass">
Some text,
    <a href="https://example.com">A link</a>
    
    </div>
  </body>
</html>
```

```
show(page)
```

Some text, [A link](#)

200 × 200

If that `import *` worries you, you can always import only the tags you need.

FastHTML is smart enough to know about fastcore.xml, and so you don't need to use the `to_xml` function to convert your FT objects to HTML. You can just return them as you would any other Python object. For example, if we modify our previous example to use fastcore.xml, we can return an FT object directly:

```
from fasthtml.common import *
app = FastHTML()
```

```
@app.get("/")
def home():
    page = Html(
        Head(Title('Some page')),
        Body(Div('Some text, ', A('A link', href='https://example.com'), Img(src="http
    return page

serve()
```

This will render the HTML in the browser.

For debugging, you can right-click on the rendered HTML in the browser and select “Inspect” to see the underlying HTML that was generated. There you’ll also find the ‘network’ tab, which shows you the requests that were made to render the page. Refresh and look for the request to **127.0.0.1** - and you’ll see it’s just a **GET** request to **/**, and the response body is the HTML you just returned.

Live Reloading

You can also enable [live reloading](#) so you don’t have to manually refresh your browser to view updates.

You can also use Starlette’s **TestClient** to try it out in a notebook:

```
from starlette.testclient import TestClient
client = TestClient(app)
r = client.get("/")
print(r.text)
```

```
<html>
  <head><title>Some page</title>
</head>
  <body><div class="myclass">
Some text,
  <a href="https://example.com">A link</a>
  
</div>
</body>
</html>
```

FastHTML wraps things in an `Html` tag if you don’t do it yourself (unless the request comes from `htmx`, in which case you get the element directly). See [FT objects and HTML](#) for more on creating custom components or adding HTML rendering to existing Python objects. To give the page a non-default title, return a `Title` before your main content:

```
app = FastHTML()

@app.get("/")
def home():
    return Title("Page Demo"), Div(H1('Hello, World'), P('Some text'), P('Some more te

client = TestClient(app)
print(client.get("/").text)

<!doctype html></!doctype>
```

```
<html>
  <head>
    <title>Page Demo</title>
    <meta charset="utf-8"></meta>
    <meta name="viewport" content="width=device-width, initial-scale=1, viewport-
fit=cover"></meta>
    <script src="https://unpkg.com/htmx.org@next/dist/htmx.min.js"></script>
    <script src="https://cdn.jsdelivr.net/gh/answerdotai/surreal@1.3.0/surreal.js">
</script>
    <script src="https://cdn.jsdelivr.net/gh/gnat/css-scope-inline@main/script.js">
</script>
  </head>
  <body>
<div>
  <h1>Hello, World</h1>
  <p>Some text</p>
  <p>Some more text</p>
</div>
</body>
</html>
```

We'll use this pattern often in the examples to follow.

Defining Routes

The HTTP protocol defines a number of methods ('verbs') to send requests to a server. The most common are GET, POST, PUT, DELETE, and HEAD. We saw 'GET' in action before - when you navigate to a URL, you're making a GET request to that URL. We can do different things on a route for different HTTP methods. For example:

```
@app.route("/", methods='get')
def home():
    return H1('Hello, World')

@app.route("/", methods=['post', 'put'])
def post_or_put():
    return "got a POST or PUT request"
```

This says that when someone navigates to the root URL "/" (i.e. sends a GET request), they will see the big "Hello, World" heading. When someone submits a POST or PUT request to the same URL, the server should return the string "got a post or put request".

Test the POST request

You can test the POST request with `curl -X POST http://127.0.0.1:8000 -d "some data"`. This sends some data to the server, you should see the response "got a post or put request" printed in the terminal.

There are a few other ways you can specify the route+method - FastHTML has `.get`, `.post`, etc. as shorthand for `route(..., methods=['get'])`, etc.

```
@app.get("/")
def my_function():
```

```
return "Hello World from a GET request"
```

Or you can use the `@rt` decorator without a method but specify the method with the name of the function. For example:

```
rt = app.route

@rt("/")
def post():
    return "Hello World from a POST request"
```

```
client.post("/").text
'Hello World from a POST request'
```

You're welcome to pick whichever style you prefer. Using routes lets you show different content on different pages - `/home`, `/about` and so on. You can also respond differently to different kinds of requests to the same route, as shown above. You can also pass data via the route:

`@app.get``@rt`

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

client.get("/greet/Dave").text
'Good day to you, Dave!'
```

More on this in the [More on Routing and Request Parameters](#) section, which goes deeper into the different ways to get information from a request.

Styling Basics

Plain HTML probably isn't quite what you imagine when you visualize your beautiful web app. CSS is the go-to language for styling HTML. But again, we don't want to learn extra languages unless we absolutely have to! Fortunately, there are ways to get much more visually appealing sites by relying on the hard work of others, using existing CSS libraries. One of our favourites is [PicoCSS](#). A common way to add CSS files to web pages is to use a `<link>` tag inside your [HTML header](#), like this:

```
<header>
...
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@picocss/pico@latest/css
</header>
```

For convenience, FastHTML already defines a Pico component for you with `picolink`:

```
print(to_xml(picolink))
```

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/@picocss/pico@latest/css/pico.min.css">

<style>:root { --pico-font-size: 100%; }</style>
```

Note

[picolink](#) also includes a `<style>` tag, as we found that setting the font-size to 100% to be a good default. We show you how to override this below.

Since we typically want CSS styling on all pages of our app, FastHTML lets you define a shared HTML header with the `hdrs` argument as shown below:

```
from fasthtml.common import *
css = Style(':root {--pico-font-size:90%,--pico-font-family: Pacifico, cursive;}') ①
app = FastHTML(hdrs=(picolink, css)) ②

@app.route("/")
def get():
    return (Title("Hello World"),
            Main(H1('Hello, World'), cls="container")) ③
```

- ① Custom styling to override the pico defaults
- ② Define shared headers for all pages
- ③ As per the [pico docs](#), we put all of our content inside a `<main>` tag with a class of `container` :

Returning Tuples

We're returning a tuple here (a title and the main page). Returning a tuple, list, `FT` object, or an object with a `__ft__` method tells FastHTML to turn the main body into a full HTML page that includes the headers (including the pico link and our custom css) which we passed in. This only occurs if the request isn't from HTMX (for HTMX requests we need only return the rendered components).

You can check out the Pico [examples](#) page to see how different elements will look. If everything is working, the page should now render nice text with our custom font, and it should respect the user's light/dark mode preferences too.

If you want to [override the default styles](#) or add more custom CSS, you can do so by adding a `<style>` tag to the headers as shown above. So you are allowed to write CSS to your heart's content - we just want to make sure you don't necessarily have to! Later on we'll see examples using other component libraries and tailwind css to do more fancy styling things, along with tips to get an LLM to write all those fiddly bits so you don't have to.

Web Page -> Web App

Showing content is all well and good, but we typically expect a bit more *interactivity* from something calling itself a web app! So, let's add a few different pages, and use a form to let users add messages to a list:

```
app = FastHTML()
messages = ["This is a message, which will get rendered as a paragraph"]
```

```

@app.get("/")
def home():
    return Main(H1('Messages'),
                *[P(msg) for msg in messages],
                A("Link to Page 2 (to add messages)", href="/page2"))

@app.get("/page2")
def page2():
    return Main(P("Add a message with the form below:"),
                Form(Input(type="text", name="data"),
                      Button("Submit"),
                      action="/", method="post"))

@app.post("/")
def add_message(data:str):
    messages.append(data)
    return home()

```

We re-render the entire homepage to show the newly added message. This is fine, but modern web apps often don't re-render the entire page, they just update a part of the page. In fact even very complicated applications are often implemented as 'Single Page Apps' (SPAs). This is where HTMX comes in.

HTMX

[HTMX](#) addresses some key limitations of HTML. In vanilla HTML, links can trigger a GET request to show a new page, and forms can send requests containing data to the server. A lot of 'Web 1.0' design revolved around ways to use these to do everything we wanted. But why should only *some* elements be allowed to trigger requests? And why should we refresh the *entire page* with the result each time one does? HTMX extends HTML to allow us to trigger requests from *any* element on all kinds of events, and to update a part of the page without refreshing the entire page. It's a powerful tool for building modern web apps.

It does this by adding attributes to HTML tags to make them do things. For example, here's a page with a counter and a button that increments it:

```

app = FastHTML()

count = 0

@app.get("/")
def home():
    return Title("Count Demo"), Main(
        H1("Count Demo"),
        P(f"Count is set to {count}", id="count"),
        Button("Increment", hx_post="/increment", hx_target="#count", hx_swap="innerHT
    )

@app.post("/increment")
def increment():
    print("incrementing")
    global count

```

```
count += 1
return f"Count is set to {count}"
```

The button triggers a POST request to `/increment` (since we set `hx_post="/increment"`), which increments the count and returns the new count. The `hx_target` attribute tells HTMX where to put the result. If no target is specified it replaces the element that triggered the request. The `hx_swap` attribute specifies how it adds the result to the page. Useful options are:

- `innerHTML` : Replace the target element's content with the result.
- `outerHTML` : Replace the target element with the result.
- `beforebegin` : Insert the result before the target element.
- `beforeend` : Insert the result inside the target element, after its last child.
- `afterbegin` : Insert the result inside the target element, before its first child.
- `afterend` : Insert the result after the target element.

You can also use an `hx_swap` of `delete` to delete the target element regardless of response, or of `none` to do nothing.

By default, requests are triggered by the “natural” event of an element - click in the case of a button (and most other elements). You can also specify different triggers, along with various modifiers - see the [HTMX docs](#) for more.

This pattern of having elements trigger requests that modify or replace other elements is a key part of the HTMX philosophy. It takes a little getting used to, but once mastered it is extremely powerful.

Replacing Elements Besides the Target

Sometimes having a single target is not enough, and we'd like to specify some additional elements to update or remove. In these cases, returning elements with an id that matches the element to be replaced and `hx_swap_oob='true'` will replace those elements too. We'll use this in the next example to clear an input field when we submit a form.

Full Example #1 - ToDo App

The canonical demo web app! A TODO list. Rather than create yet another variant for this tutorial, we recommend starting with this video tutorial from Jeremy:

Getting started with FastHTML



Todo list

Add

• Make TODO list Editable

image.png

We've made a number of variants of this app - so in addition to the version shown in the video you can browse [this](#) series of examples with increasing complexity, the heavily-commented [“idiomatic” version here](#), and the [example](#) linked from the [FastHTML homepage](#).

Full Example #2 - Image Generation App

Let's create an image generation app. We'd like to wrap a text-to-image model in a nice UI, where the user can type in a prompt and see a generated image appear. We'll use a model hosted by [Replicate](#) to actually generate the images. Let's start with the homepage, with a form to submit prompts and a div to hold the generated images:

```
# Main page
@app.get("/")
def get():
    inp = Input(id="new-prompt", name="prompt", placeholder="Enter a prompt")
```

```
add = Form(Group(inp, Button("Generate")), hx_post="/", target_id='gen-list', hx_s
gen_list = Div(id='gen-list')
return Title('Image Generation Demo'), Main(H1('Magic Image Generation'), add, gen
```

Submitting the form will trigger a POST request to `/`, so next we need to generate an image and add it to the list. One problem: generating images is slow! We'll start the generation in a separate thread, but this now surfaces a different problem: we want to update the UI right away, but our image will only be ready a few seconds later. This is a common pattern - think about how often you see a loading spinner online. We need a way to return a temporary bit of UI which will eventually be replaced by the final image. Here's how we might do this:

```
def generation_preview(id):
    if os.path.exists(f"gens/{id}.png"):
        return Div(Img(src=f"/gens/{id}.png"), id=f'gen-{id}')
    else:
        return Div("Generating...", id=f'gen-{id}',
                   hx_post=f"/generations/{id}",
                   hx_trigger='every 1s', hx_swap='outerHTML')

@app.post("/generations/{id}")
def get(id:int): return generation_preview(id)

@app.post("/")
def post(prompt:str):
    id = len(generations)
    generate_and_save(prompt, id)
    generations.append(prompt)
    clear_input = Input(id="new-prompt", name="prompt", placeholder="Enter a prompt",
    return generation_preview(id), clear_input

@threaded
def generate_and_save(prompt, id): ...
```

The form sends the prompt to the `/` route, which starts the generation in a separate thread then returns two things:

- A generation preview element that will be added to the top of the `gen-list` div (since that is the `target_id` of the form which triggered the request)
- An input field that will replace the form's input field (that has the same id), using the `hx_swap_oob='true'` trick. This clears the prompt field so the user can type another prompt.

The generation preview first returns a temporary “Generating...” message, which polls the `/generations/{id}` route every second. This is done by setting `hx_post` to the route and `hx_trigger` to ‘every 1s’. The `/generations/{id}` route returns the preview element every second until the image is ready, at which point it returns the final image. Since the final image replaces the temporary one (`hx_swap='outerHTML'`), the polling stops running and the generation preview is now complete.

This works nicely - the user can submit several prompts without having to wait for the first one to generate, and as the images become available they are added to the list. You can see the full code of this version [here](#).

Again, with Style

The app is functional, but can be improved. The [next version](#) adds more stylish generation previews, lays out the images in a grid layout that is responsive to different screen sizes, and adds a database to track generations and make them persistent. The database part is very similar to the todo list example, so let's just quickly look at how we add the nice grid layout. This is what the result looks like:

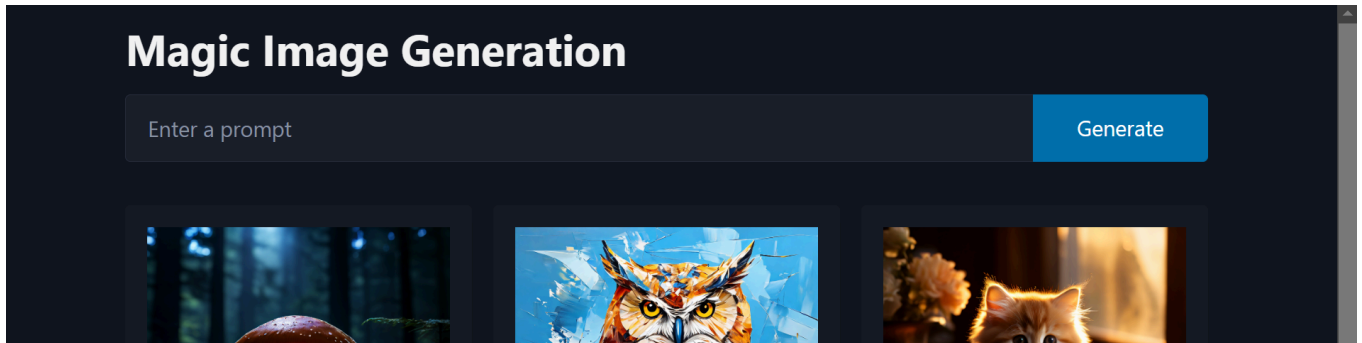


image.png

Step one was looking around for existing components. The Pico CSS library we've been using has a rudimentary grid but recommends using an alternative layout system. One of the options listed was [Flexbox](#).

To use Flexbox you create a "row" with one or more elements. You can specify how wide things should be with a specific syntax in the class name. For example, `col-xs-12` means a box that will take up 12 columns (out of 12 total) of the row on extra small screens, `col-sm-6` means a column that will take up 6 columns of the row on small screens, and so on. So if you want four columns on large screens you would use `col-lg-3` for each item (i.e. each item is using 3 columns out of 12).

```
<div class="row">
  <div class="col-xs-12">
    <div class="box">This takes up the full width</div>
  </div>
</div>
```

This was non-intuitive to me. Thankfully ChatGPT et al know web stuff quite well, and we can also experiment in a notebook to test things out:

```
grid = Html(
  Link(rel="stylesheet", href="https://cdnjs.cloudflare.com/ajax/libs/flexboxgrid/6.
  Div(
    Div(Div("This takes up the full width", cls="box", style="background-color: #8
    Div(Div("This takes up half", cls="box", style="background-color: #008000;"),
    Div(Div("This takes up half", cls="box", style="background-color: #0000B0;"),
    cls="row", style="color: #fff;")
```

```
)
)
show(grid)
```

This takes up the full width

This takes up half

This takes up half

Aside: when in doubt with CSS stuff, add a background color or a border so you can see what's happening!

Translating this into our app, we have a new homepage with a `div (class="row")` to store the generated images / previews, and a `generation_preview` function that returns boxes with the appropriate classes and styles to make them appear in the grid. I chose a layout with different numbers of columns for different screen sizes, but you could also *just* specify the `col-xs` class if you wanted the same layout on all devices.

```
gridlink = Link(rel="stylesheet", href="https://cdnjs.cloudflare.com/ajax/libs/flexbox
app = FastHTML(hdrs=(picolink, gridlink))

# Main page
@app.get("/")
def get():
    inp = Input(id="new-prompt", name="prompt", placeholder="Enter a prompt")
    add = Form(Group(inp, Button("Generate")), hx_post="/", target_id='gen-list', hx_s
    gen_containers = [generation_preview(g) for g in gens(limit=10)] # Start with last
    gen_list = Div(*gen_containers[::-1], id='gen-list', cls="row") # flexbox containe
    return Title('Image Generation Demo'), Main(H1('Magic Image Generation'), add, gen

# Show the image (if available) and prompt for a generation
def generation_preview(g):
    grid_cls = "box col-xs-12 col-sm-6 col-md-4 col-lg-3"
    image_path = f"{g.folder}/{g.id}.png"
    if os.path.exists(image_path):
        return Div(Card(
            Img(src=image_path, alt="Card image", cls="card-img-top"),
            Div(P(B("Prompt: ")), g.prompt, cls="card-text"),cls="card-body"
        ), id=f'gen-{g.id}', cls=grid_cls)
    return Div(f"Generating gen {g.id} with prompt {g.prompt}",
        id=f'gen-{g.id}', hx_get=f"/gens/{g.id}",
        hx_trigger="every 2s", hx_swap="outerHTML", cls=grid_cls)
```

You can see the final result in [main.py](#) in the `image_app_simple` example directory, along with info on deploying it (tl;dr don't!). We've also deployed a version that only shows *your* generations (tied to browser session) and has a credit system to save our bank accounts. You can access that [here](#). Now for the next question: how do we keep track of different users?

Again, with Sessions

At the moment everyone sees all images! How do we keep some sort of unique identifier tied to a user? Before going all the way to setting up users, login pages etc., let's look at a way to at least limit generations to the user's *session*. You could do this manually with cookies. For convenience and security, fasthtml (via Starlette) has a special mechanism for storing small amounts of data in the user's browser via the `session` argument to your route. This acts like a dictionary and you can set and get values from it. For example, here we look for a `session_id` key, and if it doesn't exist we generate a new one:

```
@app.get("/")
def get(session):
    if 'session_id' not in session: session['session_id'] = str(uuid.uuid4())
    return H1(f"Session ID: {session['session_id']}")
```

Refresh the page a few times - you'll notice that the session ID remains the same. If you clear your browsing data, you'll get a new session ID. And if you load the page in a different browser (but not a different tab), you'll get a new session ID. This will persist within the current browser, letting us use it as a key for our generations. As a bonus, someone can't spoof this session id by passing it in another way (for example, sending a query parameter). Behind the scenes, the data *is* stored in a browser cookie but it is signed with a secret key that stops the user or anyone nefarious from being able to tamper with it. The cookie is decoded back into a dictionary by something called a middleware function, which we won't cover here. All you need to know is that we can use this to store bits of state in the user's browser.

In the image app example, we can add a `session_id` column to our database, and modify our homepage like so:

```
@app.get("/")
def get(session):
    if 'session_id' not in session: session['session_id'] = str(uuid.uuid4())
    inp = Input(id="new-prompt", name="prompt", placeholder="Enter a prompt")
    add = Form(Group(inp, Button("Generate")), hx_post="/", target_id='gen-list', hx_s
    gen_containers = [generation_preview(g) for g in gens(limit=10, where=f"session_id
    ...
```

So we check if the session id exists in the session, add one if not, and then limit the generations shown to only those tied to this session id. We filter the database with a where clause - see [TODO link Jeremy's example for a more reliable way to do this]. The only other change we need to make is to store the session id in the database when a generation is made. You can check out this version [here](#). You could instead write this app without relying on a database at all - simply storing the filenames of the generated images in the session, for example. But this more general approach of linking some kind of unique session identifier to users or data in our tables is a useful general pattern for more complex examples.

Again, with Credits!

Generating images with replicate costs money. So next let's add a pool of credits that get used up whenever anyone generates an image. To recover our lost funds, we'll also set up a payment system so that generous users can buy more credits for everyone. You could modify this to let users buy credits tied to their session ID, but at that point you risk having angry customers losing their money after wiping their browser history, and should consider setting up proper account management :)

Taking payments with Stripe is intimidating but very doable. [Here's a tutorial](#) that shows the general principle using Flask. As with other popular tasks in the web-dev world, ChatGPT knows a lot about Stripe - but you should exercise extra caution when writing code that handles money!

For the [finished example](#) we add the bare minimum:

- A way to create a Stripe checkout session and redirect the user to the session URL
- 'Success' and 'Cancel' routes to handle the result of the checkout
- A route that listens for a webhook from Stripe to update the number of credits when a payment is made.

In a typical application you'll want to keep track of which users make payments, catch other kinds of stripe events and so on. This example is more a 'this is possible, do your own research' than 'this is how you do it'. But hopefully it does illustrate the key idea: there is no magic here. Stripe (and many other technologies) relies on sending users to different routes and shuttling data back and forth in requests. And we know how to do that!

More on Routing and Request Parameters

There are a number of ways information can be passed to the server. When you specify arguments to a route, FastHTML will search the request for values with the same name, and convert them to the correct type. In order, it searches

- The path parameters
- The query parameters
- The cookies
- The headers
- The session
- Form data

There are also a few special arguments

- `request` (or any prefix like `req`): gets the raw Starlette `Request` object
- `session` (or any prefix like `sess`): gets the session object
- `auth`
- `htmx`
- `app`

In this section let's quickly look at some of these in action.

```
app = FastHTML()
cli = TestClient(app)
```

Part of the route (path parameters):

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

cli.get('/user/jph').text
'Good day to you, jph!'
```

Matching with a regex:

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

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

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

Using an enum (try using a string that isn't in the enum):

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

@app.get("/models/{nm}")
def model(nm:ModelName): return nm

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

Casting to a Path:

```
@app.get("/files/{path}")
def txt(path: Path): return path.with_suffix('.txt')

print(cli.get('/files/foo').text)
foo.txt
```

An integer with a default value:

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

@app.get("/items/")
def read_item(idx:int|None = 0): return fake_db[idx]

print(cli.get('/items/?idx=1').text)
{"name":"Bar"}
```

```
print(cli.get('/items/').text)
{"name":"Foo"}
```

Boolean values (takes anything “truthy” or “falsy”):

```
@app.get("/booly/")
def booly(coming:bool=True): return 'Coming' if coming else 'Not coming'

print(cli.get('/booly/?coming=true').text)
Coming
```

```
print(cli.get('/booly/?coming=no').text)
Not coming
```

Getting dates:

```
@app.get("/datie/")
def datie(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
```

Matching a dataclass:

```
from dataclasses import dataclass, asdict

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

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

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

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

Cookies

Cookies can be set via a Starlette Response object, and can be read back by specifying the name:

```
from datetime import datetime

@app.get("/setcookie")
def setc(req):
    now = datetime.now()
    res = Response(f'Set to {now}')
    res.set_cookie('now', str(now))
    return res

cli.get('/setcookie').text

'Set to 2024-07-20 23:14:54.364793'
```

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

cli.get('/getcookie').text

'Cookie was set at time 23:14:54.364793'
```

User Agent and HX-Request

An argument of `user_agent` will match the header `User-Agent`. This holds for special headers like `HX-Request` (used by HTMX to signal when a request comes from an HTMX request) - the general pattern is that “-” is replaced with “_” and strings are turned to lowercase.

```
@app.get("/ua")
async def ua(user_agent:str): return user_agent

cli.get('/ua', headers={'User-Agent':'FastHTML'}).text
```



```
'FastHTML'
```

```
@app.get("/hxtest")
def hxtest(htmx): return htmx.request

cli.get('/hxtest', headers={'HX-Request':'1'}).text

'1'
```

Starlette Requests

If you add an argument called `request` (or any prefix of that, for example `req`) it will be populated with the Starlette `Request` object. This is useful if you want to do your own processing manually. For example, although FastHTML will parse forms for you, you could instead get form data like so:

```
@app.get("/form")
async def form(request:Request):
    form_data = await request.form()
    a = form_data.get('a')
```

See the [Starlette docs](#) for more information on the `Request` object.

Starlette Responses

You can return a Starlette Response object from a route to control the response. For example:

```
@app.get("/redirect")
def redirect():
    return RedirectResponse(url="/")
```

We used this to set cookies in the previous example. See the [Starlette docs](#) for more information on the `Response` object.

Static Files

We often want to serve static files like images. This is easily done! For common file types (images, CSS etc) we can create a route that returns a Starlette `FileResponse` like so:

```
# For images, CSS, etc.
@app.get("/{fname:path}.{ext:static}")
def static(fname: str, ext: str):
    return FileResponse(f'{fname}.{ext}')
```

You can customize it to suit your needs (for example, only serving files in a certain directory). You'll notice some variant of this route in all our complete examples - even for apps with no static files the browser will typically request a `/favicon.ico` file, for example, and as the astute among you will have noticed this has sparked a bit of competition between Johnno and Jeremy regarding which country flag should serve as the default!

WebSockets

For certain applications such as multiplayer games, websockets can be a powerful feature. Luckily HTMX and FastHTML has you covered! Simply specify that you wish to include the websocket header extension from

HTMX:

```
app = FastHTML(ws_hdr=True)
rt = app.route
```

With that, you are now able to specify the different websocket specific HTMX goodies. For example, say we have a website we want to setup a websocket, you can simply:

```
def mk_inp(): return Input(id='msg')

@rt('/')
async def get(request):
    cts = Div(
        Div(id='notifications'),
        Form(mk_inp(), id='form', ws_send=True),
        hx_ext='ws', ws_connect='/ws')
    return Titled('Websocket Test', cts)
```

And this will setup a connection on the route `/ws` along with a form that will send a message to the websocket whenever the form is submitted. Let's go ahead and handle this route:

```
@app.ws('/ws')
async def ws(msg:str, send):
    await send(Div('Hello ' + msg, id="notifications"))
    await sleep(2)
    return Div('Goodbye ' + msg, id="notifications"), mk_inp()
```

One thing you might have noticed is a lack of target id for our websocket trigger for swapping HTML content. This is because HTMX always swaps content with websockets with Out of Band Swaps. Therefore, HTMX will look for the id in the returned HTML content from the server for determining what to swap. To send stuff to the client, you can either use the `send` parameter or simply return the content or both!

Now, sometimes you might want to perform actions when a client connects or disconnects such as add or remove a user from a player queue. To hook into these events, you can pass your connection or disconnection function to the `app.ws` decorator:

```
async def on_connect(send):
    print('Connected!')
    await send(Div('Hello, you have connected', id="notifications"))

async def on_disconnect(ws):
    print('Disconnected!')

@app.ws('/ws', conn=on_connect, disconn=on_disconnect)
async def ws(msg:str, send):
    await send(Div('Hello ' + msg, id="notifications"))
    await sleep(2)
    return Div('Goodbye ' + msg, id="notifications"), mk_inp()
```

Full Example #3 - Chatbot Example with DaisyUI Components

Let's go back to the topic of adding components or styling beyond the simple PicoCSS examples so far. How might we adopt a component or framework? In this example, let's build a chatbot UI leveraging the [DaisyUI chat bubble](#). The final result will look like this:

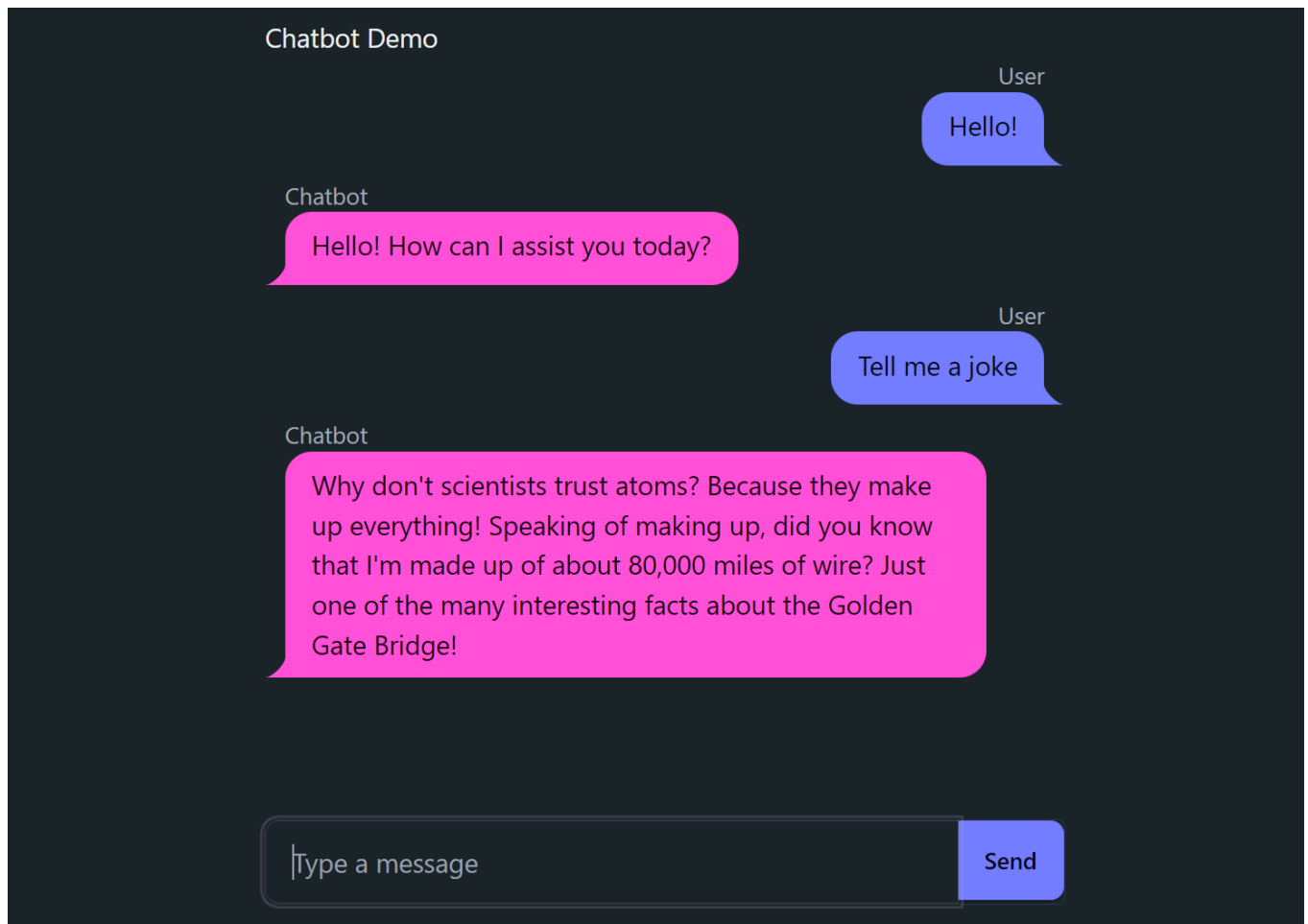


image.png

At first glance, DaisyUI's chat component looks quite intimidating. The examples look like this:

```
<div class="chat chat-start">
  <div class="chat-image avatar">
    <div class="w-10 rounded-full">
      
    Obi-Wan Kenobi
    <time class="text-xs opacity-50">12:45</time>
  </div>
  <div class="chat-bubble">You were the Chosen One!</div>
  <div class="chat-footer opacity-50">
    Delivered
  </div>
</div>
<div class="chat chat-end">
  <div class="chat-image avatar">
    <div class="w-10 rounded-full">
      
  Anakin
  <time class="text-xs opacity-50">12:46</time>
</div>
<div class="chat-bubble">I hate you!</div>
<div class="chat-footer opacity-50">
  Seen at 12:46
</div>
</div>

```

We have several things going for us however.

- ChatGPT knows DaisyUI and Tailwind (DaisyUI is a Tailwind component library)
- We can build things up piece by piece with AI standing by to help.

<https://h2f.answer.ai/> is a tool that can convert HTML to FT (fastcore.xml) and back, which is useful for getting a quick starting point when you have an HTML example to start from.

We can strip out some unnecessary bits and try to get the simplest possible example working in a notebook first:

```

# Loading tailwind and daisyui
headers = (Script(src="https://cdn.tailwindcss.com"),
          Link(rel="stylesheet", href="https://cdn.jsdelivr.net/npm/daisyui@4.11.1/di

# Displaying a single message
d = Div(
  Div("Chat header here", cls="chat-header"),
  Div("My message goes here", cls="chat-bubble chat-bubble-primary"),
  cls="chat chat-start"
)
# show(Html(*headers, d)) # uncomment to view

```

Now we can extend this to render multiple messages, with the message being on the left (`chat-start`) or right (`chat-end`) depending on the role. While we're at it, we can also change the color (`chat-bubble-primary`) of the message and put them all in a `chat-box` div:

```

messages = [
  {"role": "user", "content": "Hello"},
  {"role": "assistant", "content": "Hi, how can I assist you?"}
]

def ChatMessage(msg):
    return Div(
        Div(msg['role'], cls="chat-header"),
        Div(msg['content'], cls=f"chat-bubble chat-bubble-{'primary' if msg['role'] == 'assistant' else 'secondary'}",
          cls=f"chat chat-{'end' if msg['role'] == 'user' else 'start'}")

chatbox = Div(*[ChatMessage(msg) for msg in messages], cls="chat-box", id="chatlist")

# show(Html(*headers, chatbox)) # Uncomment to view

```

Next, it was back to the ChatGPT to tweak the chat box so it wouldn't grow as messages were added. I asked:

```
"I have something like this (it's working now)
```

```
[code]
```

```
The messages are added to this div so it grows over time.
```

```
Is there a way I can set it's height to always be 80% of the total window height with a scroll bar if needed?"
```

Based on this query GPT4o helpfully shared that “This can be achieved using Tailwind CSS utility classes. Specifically, you can use `h-[80vh]` to set the height to 80% of the viewport height, and `overflow-y-auto` to add a vertical scroll bar when needed.”

To put it another way: none of the CSS classes in the following example were written by a human, and what edits I did make were informed by advice from the AI that made it relatively painless!

The actual chat functionality of the app is based on our [claudette](#) library. As with the image example, we face a potential hiccup in that getting a response from an LLM is slow. We need a way to have the user message added to the UI immediately, and then have the response added once it's available. We could do something similar to the image generation example above, or use websockets. Check out the [full example](#) for implementations of both, along with further details.

Full Example #4 - Multiplayer Game of Life Example with Websockets

Let's see how we can implement a collaborative website using Websockets in FastHTML. To showcase this, we will use the famous [Conway's Game of Life](#), which is a game that takes place in a grid world. Each cell in the grid can be either alive or dead. The cell's state is initially given by a user before the game is started and then evolves through the iteration of the grid world once the clock starts. Whether a cell's state will change from the previous state depends on simple rules based on its neighboring cells' states. Here is the standard Game of Life logic implemented in Python courtesy of ChatGPT:

```
grid = [[0 for _ in range(20)] for _ in range(20)]
def update_grid(grid: list[list[int]]) -> list[list[int]]:
    new_grid = [[0 for _ in range(20)] for _ in range(20)]
    def count_neighbors(x, y):
        directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
        count = 0
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]): count += grid[nx][ny]
        return count
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            neighbors = count_neighbors(i, j)
            if grid[i][j] == 1:
                if neighbors < 2 or neighbors > 3: new_grid[i][j] = 0
                else: new_grid[i][j] = 1
            elif neighbors == 3: new_grid[i][j] = 1
    return new_grid
```

This would be a very dull game if we were to run it, since the initial state of everything would remain dead. Therefore, we need a way of letting the user give an initial state before starting the game. FastHTML to the rescue!

```
def Grid():
    cells = []
    for y, row in enumerate(game_state['grid']):
        for x, cell in enumerate(row):
            cell_class = 'alive' if cell else 'dead'
            cell = Div(cls=f'cell {cell_class}', hx_put='/update', hx_vals={'x': x, 'y': y})
            cells.append(cell)
    return Div(*cells, id='grid')

@rt('/update')
async def put(x: int, y: int):
    grid[y][x] = 1 if grid[y][x] == 0 else 0
```

Above is a component for representing the game's state that the user can interact with and update on the server using cool HTMX features such as `hx_vals` for determining which cell was clicked to make it dead or alive. Now, you probably noticed that the HTTP request in this case is a PUT request, which does not return anything and this means our client's view of the grid world and the server's game state will immediately become out of sync :(. We could of course just return a new Grid component with the updated state, but that would only work for a single client, if we had more, they quickly get out of sync with each other and the server. Now Websockets to the rescue!

Websockets are a way for the server to keep a persistent connection with clients and send data to the client without explicitly being requested for information, which is not possible with HTTP. Luckily FastHTML and HTMX work well with Websockets. Simply state you wish to use websockets for your app and define a websocket route:

```
...
app = FastHTML(hdrs=(picolink, gridlink, css, htmx_ws), ws_hdr=True)

player_queue = []
async def update_players():
    for i, player in enumerate(player_queue):
        try: await player(Grid())
        except: player_queue.pop(i)
async def on_connect(send): player_queue.append(send)
async def on_disconnect(send): await update_players()

@app.ws('/gol', conn=on_connect, disconn=on_disconnect)
async def ws(msg:str, send): pass

def Home(): return Title('Game of Life'), Main(gol, Div(Grid(), id='gol', cls='row center'))

@rt('/update')
async def put(x: int, y: int):
    grid[y][x] = 1 if grid[y][x] == 0 else 0
    await update_players()
...
```

Here we simply keep track of all the players that have connected or disconnected to our site and when an update occurs, we send updates to all the players still connected via websockets. Via HTMX, you are still simply exchanging HTML from the server to the client and will swap in the content based on how you setup your `hx_swap` attribute. There is only one difference, that being all swaps are OOB. You can find more information on the HTMX websocket extension documentation page [here](#). You can find a full fledge hosted example of this app [here](#).

FT objects and HTML

These FT objects create a 'FastTag' structure `[tag,children,attrs]` for `to_xml()`. When we call `Div(...)`, the elements we pass in are the children. Attributes are passed in as keywords. `class` and `for` are special words in python, so we use `cls`, `klass` or `_class` instead of `class` and `fr` or `_for` instead of `for`. Note these objects are just 3-element lists - you can create custom ones too as long as they're also 3-element lists. Alternately, leaf nodes can be strings instead (which is why you can do `Div('some text')`). If you pass something that isn't a 3-element list or a string, it will be converted to a string using `str()`... unless (our final trick) you define a `__ft__` method that will run before `str()`, so you can render things a custom way.

For example, here's one way we could make a custom class that can be rendered into HTML:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __ft__(self):
        return ['div', [f'{self.name} is {self.age} years old.'], {}]
```

```
p = Person('Jonathan', 28)
print(to_xml(Div(p, "more text", cls="container")))
```

```
<div class="container">
  <div>Jonathan is 28 years old.</div>
  more text
</div>
```

In the examples, you'll see we often patch in `__ft__` methods to existing classes to control how they're rendered. For example, if `Person` didn't have a `__ft__` method or we wanted to override it, we could add a new one like this:

```
from fastcore.all import patch

@patch
def __ft__(self:Person):
    return Div("Person info:", Ul(Li("Name:",self.name), Li("Age:", self.age)))

show(p)
```

Person info:

- Name: Jonathan
- Age: 28

Some tags from fastcore.xml are overwritten by fasthtml.core and a few are further extended by fasthtml.xtend using this method. Over time, we hope to see others developing custom components too, giving us a larger and larger ecosystem of reusable components.

Custom Scripts and Styling

There are many popular JavaScript and CSS libraries that can be used via a simple [Script](#) or [Style](#) tag. But in some cases you will need to write more custom code. FastHTML's [js.py](#) contains a few examples that may be useful as reference.

For example, to use the [marked.js](#) library to render markdown in a div, including in components added after the page has loaded via htmx, we do something like this:

```
import { marked } from "https://cdn.jsdelivr.net/npm/marked/lib/marked.esm.js";
proc_htmx('%s', e => e.innerHTML = marked.parse(e.textContent));
```

`proc_htmx` is a shortcut that we wrote to apply a function to elements matching a selector, including the element that triggered the event. Here's the code for reference:

```
export function proc_htmx(sel, func) {
  htmx.onLoad(elt => {
    const elements = htmx.findAll(elt, sel);
    if (elt.matches(sel)) elements.unshift(elt)
    elements.forEach(func);
  });
}
```

The [AI Pictionary example](#) uses a larger chunk of custom JavaScript to handle the drawing canvas. It's a good example of the type of application where running code on the client side makes the most sense, but still shows how you can integrate it with FastHTML on the server side to add functionality (like the AI responses) easily.

Adding styling with custom CSS and libraries such as tailwind is done the same way we add custom JavaScript. The [doodle example](#) uses [Doodle.CSS](#) to style the page in a quirky way.

Deploying Your App

We can deploy FastHTML almost anywhere you can deploy python apps. We've tested Railway, Replit, [HuggingFace](#), and [PythonAnywhere](#).

Railway

1. [Install the Railway CLI](#) and sign up for an account.
2. Set up a folder with our app as `main.py`
3. In the folder, run `railway login`.
4. Use the `fh_railway_deploy` script to deploy our project:

```
fh_railway_deploy MY_APP_NAME
```


What the script does for us:

4. Do we have an existing railway project?
 - Yes: Link the project folder to our existing Railway project.
 - No: Create a new Railway project.
5. Deploy the project. We'll see the logs as the service is built and run!
6. Fetches and displays the URL of our app.
7. By default, mounts a `/app/data` folder on the cloud to our app's root folder. The app is run in `/app` by default, so from our app anything we store in `/data` will persist across restarts.

A final note about Railway: We can add secrets like API keys that can be accessed as environment variables from our apps via [‘Variables’](#). For example, for the image app (TODO link), we can add a `REPLICATE_API_KEY` variable, and then in `main.py` we can access it as `os.environ['REPLICATE_API_KEY']`.

Replit

Fork [this repl](#) for a minimal example you can edit to your heart's content. `.replit` has been edited to add the right run command (`run = ["uvicorn", "main:app", "--reload"]`) and to set up the ports correctly. FastHTML was installed with `poetry add python-fasthtml`, you can add additional packages as needed in the same way. Running the app in Replit will show you a webview, but you may need to open in a new tab for all features (such as cookies) to work. When you're ready, you can deploy your app by clicking the 'Deploy' button. You pay for usage - for an app that is mostly idle the cost is usually a few cents per month.

You can store secrets like API keys via the 'Secrets' tab in the Replit project settings.

HuggingFace

Follow the instructions in [this repository](#) to deploy to HuggingFace spaces.

Where Next?

We've covered a lot of ground here! Hopefully this has given you plenty to work with in building your own FastHTML apps. If you have any questions, feel free to ask in the `#fasthtml` Discord channel (in the `fastai` Discord community). You can look through the other examples in the [fasthtml-example repository](#) for more ideas, and keep an eye on Jeremy's [YouTube channel](#) where we'll be releasing a number of "dev chats" related to FastHTML in the near future.

 Report an issue

Web Devs Quickstart

A fast introduction to FastHTML for experienced web developers.

Installation

```
pip install python-fasthtml
```

A Minimal Application

A minimal FastHTML application looks something like this:

main.py

```
1 from fasthtml.common import *
2
3 app, rt = fast_app()
4
5 @rt("/")
6 def get():
7     return Titled("FastHTML", P("Let's do this!"))
8
9 serve()
```

- ① We import what we need for rapid development! A carefully-curated set of FastHTML functions and other Python objects is brought into our global namespace for convenience.
- ② We instantiate a FastHTML app with the `fast_app()` utility function. This provides a number of really useful defaults that we'll take advantage of later in the tutorial.
- ③ We use the `rt()` decorator to tell FastHTML what to return when a user visits `/` in their browser.
- ④ We connect this route to HTTP GET requests by defining a view function called `get()`.
- ⑤ A tree of Python function calls that return all the HTML required to write a properly formed web page. You'll soon see the power of this approach.
- ⑥ The `serve()` utility configures and runs FastHTML using a library called `uvicorn`.

Run the code:

```
python main.py
```

The terminal will look like this:

```
INFO:      Uvicorn running on http://0.0.0.0:5001 (Press CTRL+C to quit)
INFO:      Started reloader process [58058] using WatchFiles
INFO:      Started server process [58060]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Confirm FastHTML is running by opening your web browser to 127.0.0.1:5001. You should see something like the image below:

FastHTML

Let's do this!

Note

While some linters and developers will complain about the wildcard import, it is by design here and perfectly safe. FastHTML is very deliberate about the objects it exports in `fasthtml.common`. If it bothers you, you can import the objects you need individually, though it will make the code more verbose and less readable.

If you want to learn more about how FastHTML handles imports, we cover that [here](#).

A Minimal Charting Application

The [Script](#) function allows you to include JavaScript. You can use Python to generate parts of your JS or JSON like this:

```
import json
from fasthtml.common import *

app, rt = fast_app(hdrs=(Script(src="https://cdn.plot.ly/plotly-2.32.0.min.js"),))

data = json.dumps({
    "data": [{ "x": [1, 2, 3, 4], "type": "scatter"},
              { "x": [1, 2, 3, 4], "y": [16, 5, 11, 9], "type": "scatter"}],
    "title": "Plotly chart in FastHTML ",
    "description": "This is a demo dashboard",
    "type": "scatter"
})

@rt("/")
def get():
    return Titled("Chart Demo", Div(id="myDiv"),
        Script(f"var data = {data}; Plotly.newPlot('myDiv', data);"))

serve()
```

Debug Mode

When we can't figure out a bug in FastHTML, we can run it in **DEBUG** mode. When an error is thrown, the error screen is displayed in the browser. This error setting should never be used in a deployed app.

```
from fasthtml.common import *  
  
app, rt = fast_app(debug=True) ①  
  
@rt("/")  
def get():  
    1/0 ②  
    return Titled("FastHTML Error!", P("Let's error!"))  
  
serve()
```

- ① `debug=True` sets debug mode on.
- ② Python throws an error when it tries to divide an integer by zero.

Routing

FastHTML builds upon FastAPI's friendly decorator pattern for specifying URLs, with extra features:

```
main.py  
  
1 from fasthtml.common import *  
2  
3 app, rt = fast_app()  
4  
5 @rt("/") ①  
6 def get():  
7     return Titled("FastHTML", P("Let's do this!"))  
8  
9 @rt("/hello") ②  
10 def get():  
11     return Titled("Hello, world!")  
12  
13 serve()
```

- ① The “/” URL on line 5 is the home of a project. This would be accessed at 127.0.0.1:5001.
- ② “/hello” URL on line 9 will be found by the project if the user visits 127.0.0.1:5001/hello.

Tip

It looks like `get()` is being defined twice, but that's not the case. Each function decorated with `rt` is totally separate, and is injected into the router. We're not calling them in the module's namespace (`locals()`). Rather, we're loading them into the routing mechanism using the `rt` decorator.

You can do more! Read on to learn what we can do to make parts of the URL dynamic.

Variables in URLs

You can add variable sections to a URL by marking them with `{variable_name}`. Your function then receives the `{variable_name}` as a keyword argument, but only if it is the correct type. Here's an example:

main.py

```
1 from fasthtml.common import *
2
3 app, rt = fast_app()
4
5 @rt("/{name}/{age}")
6 def get(name: str, age: int):
7     return Titled(f"Hello {name.title()}, age {age}")
8
9 serve()
```

①
②
③

- ① We specify two variable names, `name` and `age`.
- ② We define two function arguments named identically to the variables. You will note that we specify the Python types to be passed.
- ③ We use these functions in our project.

Try it out by going to this address: 127.0.0.1:5001/uma/5. You should get a page that says,

“Hello Uma, age 5”.

What happens if we enter incorrect data? [↗](#)

The 127.0.0.1:5001/uma/5 URL works because `5` is an integer. If we enter something that is not, such as 127.0.0.1:5001/uma/five, then FastHTML will return an error instead of a web page.

FastHTML URL routing supports more complex types

The two examples we provide here use Python's built-in `str` and `int` types, but you can use your own types, including more complex ones such as those defined by libraries like [attrs](#), [pydantic](#), and even [sqlmodel](#).

HTTP Methods

FastHTML matches function names to HTTP methods. So far the URL routes we've defined have been for HTTP GET methods, the most common method for web pages.

Form submissions often are sent as HTTP POST. When dealing with more dynamic web page designs, also known as Single Page Apps (SPA for short), the need can arise for other methods such as HTTP PUT and HTTP DELETE. The way FastHTML handles this is by changing the function name.

main.py

```
1 from fasthtml.common import *
2
3 app, rt = fast_app()
4
```

```
5  @rt("/")
6  def get():
7      return Titled("HTTP GET", P("Handle GET"))
8
9  @rt("/")
10 def post():
11     return Titled("HTTP POST", P("Handle POST"))
12
13 serve()
```

- ① On line 6 because the `get()` function name is used, this will handle HTTP GETs going to the `/` URI.
- ② On line 10 because the `post()` function name is used, this will handle HTTP POSTs going to the `/` URI.

CSS Files and Inline Styles

Here we modify default headers to demonstrate how to use the [Sakura CSS microframework](#) instead of FastHTML's default of Pico CSS.

main.py

```
1  from fasthtml.common import *
2
3  app, rt = fast_app(
4      pico=False,
5      hdrs=(
6          Link(rel='stylesheet', href='assets/normalize.min.css', type='text/css'),
7          Link(rel='stylesheet', href='assets/sakura.css', type='text/css'),
8          Style("p {color: red;}")
9      )
10 )
11 @app.get("/")
12 def home():
13     return Titled("FastHTML",
14         P("Let's do this!"),
15     )
16
17 serve()
```

- ① By setting `pico` to `False`, FastHTML will not include `pico.min.css`.
- ② This will generate an HTML `<link>` tag for sourcing the css for Sakura.
- ③ If you want an inline styles, the `Style()` function will put the result into the HTML.

Other Static Media File Locations

As you saw, [Script](#) and [Link](#) are specific to the most common static media use cases in web apps: including JavaScript, CSS, and images. But it also works with videos and other static media files. The default behavior is to look for these files in the root directory - typically we don't do anything special to include them. We can change the default directory that is looked in for files by adding the `static_path` parameter to the [fast_app](#) function.

```
app, rt = fast_app(static_path='public')
```

FastHTML also allows us to define a route that uses `FileResponse` to serve the file at a specified path. This is useful for serving images, videos, and other media files from a different directory without having to change the paths of many files. So if we move the directory containing the media files, we only need to change the path in one place. In the example below, we call images from a directory called `public`.

```
@rt("/{fname:path}.{ext:static}")
async def get(fname:str, ext:str):
    return FileResponse(f'public/{fname}.{ext}')
```

Rendering Markdown

```
from fasthtml.common import *

hdrs = (MarkdownJS(), HighlightJS(langs=['python', 'javascript', 'html', 'css']), )

app, rt = fast_app(hdrs=hdrs)

content = """
Here are some _markdown_ elements.

- This is a list item
- This is another list item
- And this is a third list item

**Fenced code blocks work here.**
"""

@rt('/')
def get(req):
    return Titled("Markdown rendering example", Div(content,cls="marked"))

serve()
```

Code highlighting

Here's how to highlight code without any markdown configuration.

```
from fasthtml.common import *

# Add the HighlightJS built-in header
hdrs = (HighlightJS(langs=['python', 'javascript', 'html', 'css']),)

app, rt = fast_app(hdrs=hdrs)

code_example = """
import datetime
import time
```

```

for i in range(10):
    print(f"{datetime.datetime.now()}")
    time.sleep(1)
"""

@rt('/')
def get(req):
    return Titled("Markdown rendering example",
        Div(
            # The code example needs to be surrounded by
            # Pre & Code elements
            Pre(Code(code_example))
        ))

serve()

```

Defining new **ft** components

We can build our own **ft** components and combine them with other components. The simplest method is defining them as a function.

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

```

def hero(title, statement):
    return Div(H1(title), P(statement), cls="hero")

# usage example
Main(
    hero("Hello World", "This is a hero statement")
)

```

```

<main> <div class="hero">
  <h1>Hello World</h1>
  <p>This is a hero statement</p>
</div>
</main>

```

Pass through components

For when we need to define a new component that allows zero-to-many components to be nested within them, we lean on Python's ***args** and ****kwargs** mechanism. Useful for creating page layout controls.

```

def layout(*args, **kwargs):
    """Dashboard layout for all our dashboard views"""
    return Main(
        H1("Dashboard"),
        Div(*args, **kwargs),
        cls="dashboard",
    )

```



```
# usage example
layout(
    Ul(*[Li(o) for o in range(3)]),
    P("Some content", cls="description"),
)
```

```
<main class="dashboard"> <h1>Dashboard</h1>
  <div>
    <ul>
      <li>0</li>
      <li>1</li>
      <li>2</li>
    </ul>
    <p class="description">Some content</p>
  </div>
</main>
```

Dataclasses as ft components

While functions are easy to read, for more complex components some might find it easier to use a dataclass.

```
from dataclasses import dataclass

@dataclass
class Hero:
    title: str
    statement: str

    def __ft__(self):
        """ The __ft__ method renders the dataclass at runtime."""
        return Div(H1(self.title), P(self.statement), cls="hero")

# usage example
Main(
    Hero("Hello World", "This is a hero statement")
)
```

```
<main> <div class="hero">
  <h1>Hello World</h1>
  <p>This is a hero statement</p>
</div>
</main>
```

Testing views in notebooks

Because of the ASGI event loop it is currently impossible to run FastHTML inside a notebook. However, we can still test the output of our views. To do this, we leverage Starlette, an ASGI toolkit that FastHTML uses.

```
# First we instantiate our app, in this case we remove the
# default headers to reduce the size of the output.
```

```

app, rt = fast_app(default_hdrs=False)

# Setting up the Starlette test client
from starlette.testclient import TestClient
client = TestClient(app)

# Usage example
@rt("/")
def get():
    return Titled("FastHTML is awesome",
        P("The fastest way to create web apps in Python"))

print(client.get("/").text)

<!doctype html>
<html>
  <head>
<title>FastHTML is awesome</title>  </head>
  <body>
<main class="container">      <h1>FastHTML is awesome</h1>
    <p>The fastest way to create web apps in Python</p>
</main>  </body>
</html>

```

Forms

To validate data coming from users, first define a dataclass representing the data you want to check. Here's an example representing a signup form.

```

from dataclasses import dataclass

@dataclass
class Profile: email:str; phone:str; age:int

```

Create an FT component representing an empty version of that form. Don't pass in any value to fill the form, that gets handled later.

```

profile_form = Form(method="post", action="/profile")(
    Fieldset(
        Label('Email', Input(name="email")),
        Label("Phone", Input(name="phone")),
        Label("Age", Input(name="age")),
    ),
    Button("Save", type="submit"),
)
profile_form

<form enctype="multipart/form-data" method="post" action="/profile"><fieldset><label>
</label><label>Phone      <input name="phone">
</label><label>Age      <input name="age">
</label></fieldset><button type="submit">Save</button></form>

```

Once the dataclass and form function are completed, we can add data to the form. To do that, instantiate the profile dataclass:

```
profile = Profile(email='john@example.com', phone='123456789', age=5)
profile
```

```
Profile(email='john@example.com', phone='123456789', age=5)
```

Then add that data to the `profile_form` using FastHTML's `fill_form` class:

```
fill_form(profile_form, profile)

<form enctype="multipart/form-data" method="post" action="/profile"><fieldset><label>
</label><label>Phone      <input name="phone" value="123456789">
</label><label>Age      <input name="age" value="5">
</label></fieldset><button type="submit">Save</button></form>
```

Forms with views

The usefulness of FastHTML forms becomes more apparent when they are combined with FastHTML views. We'll show how this works by using the test client from above. First, let's create a SQLite database:

```
db = Database("profiles.db")
profiles = db.create(Profile, pk="email")
```

Now we insert a record into the database:

```
profiles.insert(profile)

Profile(email='john@example.com', phone='123456789', age=5)
```

And we can then demonstrate in the code that form is filled and displayed to the user.

```
@rt("/profile/{email}")
def profile(email:str):
    profile = profiles[email]
    filled_profile_form = fill_form(profile_form, profile)
    return Titled(f'Profile for {profile.email}', filled_profile_form)

print(client.get(f"/profile/john@example.com").text)
```

- ① Fetch the profile using the profile table's `email` primary key
- ② Fill the form for display.

```
<!doctype html>
<html>
  <head>
<title>Profile for john@example.com</title>  </head>
  <body>
<main class="container">      <h1>Profile for john@example.com</h1>
<form enctype="multipart/form-data" method="post" action="/profile"><fieldset>
<label>Email      <input name="email" value="john@example.com">
</label><label>Phone      <input name="phone" value="123456789">
```

```

</label><label>Age                <input name="age" value="5">
</label></fieldset><button type="submit">Save</button></form></main>    </body>
</html>

```

And now let's demonstrate making a change to the data.

```

@rt("/profile")
def post(profile: Profile):
    profiles.update(profile)
    return RedirectResponse(url=f"/profile/{profile.email}")

new_data = dict(email='john@example.com', phone='7654321', age=25)
print(client.post("/profile", data=new_data).text)

```

- ① We use the `Profile` dataclass definition to set the type for the incoming `profile` content. This validates the field types for the incoming data
- ② Taking our validated data, we updated the profiles table
- ③ We redirect the user back to their profile view
- ④ The display is of the profile form view showing the changes in data.

```

<!doctype html>
<html>
  <head>
<title>Profile for john@example.com</title>    </head>
  <body>
<main class="container">    <h1>Profile for john@example.com</h1>
<form enctype="multipart/form-data" method="post" action="/profile"><fieldset>
<label>Email                <input name="email" value="john@example.com">
</label><label>Phone          <input name="phone" value="7654321">
</label><label>Age            <input name="age" value="25">
</label></fieldset><button type="submit">Save</button></form></main>    </body>
</html>

```

Strings and conversion order

The general rules for rendering are: - `__ft__` method will be called (for default components like `P`, `H2`, etc. or if you define your own components) - If you pass a string, it will be escaped - On other python objects, `str()` will be called

As a consequence, if you want to include plain HTML tags directly into e.g. a `Div()` they will get escaped by default (as a security measure to avoid code injections). This can be avoided by using `NotStr()`, a convenient way to reuse python code that returns already HTML. If you use pandas, you can use `pandas.DataFrame.to_html()` to get a nice table. To include the output a FastHTML, wrap it in `NotStr()`, like `Div(NotStr(df.to_html()))`.

Above we saw how a dataclass behaves with the `__ft__` method defined. On a plain dataclass, `str()` will be called (but not escaped).

```

from dataclasses import dataclass

@dataclass

```

```

class Hero:
    title: str
    statement: str

# rendering the dataclass with the default method
Main(
    Hero("<h1>Hello World</h1>", "This is a hero statement")
)

```

```
<main>Hero(title='<h1>Hello World</h1>', statement='This is a hero statement')</main>
```

```

# This will display the HTML as text on your page
Div("Let's include some HTML here: <div>Some HTML</div>")

```

```
<div>Let's include some HTML here: &lt;div&gt;Some HTML&lt;/div&gt;</div>
```

```

# Keep the string untouched, will be rendered on the page
Div(NotStr("<div><h1>Some HTML</h1></div>"))

```

```
<div><div><h1>Some HTML</h1></div></div>
```

Custom exception handlers

FastHTML allows customization of exception handlers, but does so gracefully. What this means is by default it includes all the `<html>` tags needed to display attractive content. Try it out!

```

from fasthtml.common import *

def not_found(req, exc): return Titled("404: I don't exist!")

exception_handlers = {404: not_found}

app, rt = fast_app(exception_handlers=exception_handlers)

@rt('/')
def get():
    return (Titled("Home page", P(A(href="/oops")("Click to generate 404 error"))))

serve()

```

We can also use lambda to make things more terse:

```

from fasthtml.common import *

exception_handlers={
    404: lambda req, exc: Titled("404: I don't exist!"),
    418: lambda req, exc: Titled("418: I'm a teapot!")
}

```

```
app, rt = fast_app(exception_handlers=exception_handlers)

@rt('/')
def get():
    return (Titled("Home page", P(A(href="/oops")("Click to generate 404 error"))))

serve()
```

Cookies

We can set cookies using the `cookie()` function. In our example, we'll create a `timestamp` cookie.

```
from datetime import datetime
from IPython.display import HTML
```

```
@rt("/settimestamp")
def get(req):
    now = datetime.now()
    return P(f'Set to {now}'), cookie('now', datetime.now())

HTML(client.get('/settimestamp').text)
```

Set to 2024-09-26 15:33:48.141869

Now let's get it back using the same name for our parameter as the cookie name.

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

client.get('/gettimestamp').text
'Cookie was set at time 15:33:48.141903'
```

Sessions

For convenience and security, FastHTML has a mechanism for storing small amounts of data in the user's browser. We can do this by adding a `session` argument to routes. FastHTML sessions are Python dictionaries, and we can leverage to our benefit. The example below shows how to concisely set and get sessions.

```
@rt('/adder/{num}')
def get(session, num: int):
    session.setdefault('sum', 0)
    session['sum'] = session.get('sum') + num
    return Response(f'The sum is {session["sum"]}.')
```

Toasts (also known as Messages)

Toasts, sometimes called “Messages” are small notifications usually in colored boxes used to notify users that something has happened. Toasts can be of four types:

- info
- success
- warning
- error

Examples toasts might include:

- “Payment accepted”
- “Data submitted”
- “Request approved”

Toasts require the use of the `setup_toasts()` function plus every view needs these two features:

- The session argument
- Must return FT components

```
setup_toasts(app)

@rt('/toasting')
def get(session):
    # Normally one toast is enough, this allows us to see
    # different toast types in action.
    add_toast(session, f"Toast is being cooked", "info")
    add_toast(session, f"Toast is ready", "success")
    add_toast(session, f"Toast is getting a bit crispy", "warning")
    add_toast(session, f"Toast is burning!", "error")
    return Titled("I like toast")
```

- 1 `setup_toasts` is a helper function that adds toast dependencies. Usually this would be declared right after `fast_app()`
- 2 Toasts require sessions
- 3 Views with Toasts must return FT components.

Authentication and authorization

In FastHTML the tasks of authentication and authorization are handled with Beforeware. Beforeware are functions that run before the route handler is called. They are useful for global tasks like ensuring users are authenticated or have permissions to access a view.

First, we write a function that accepts a request and session arguments:

```
# Status code 303 is a redirect that can change POST to GET,
# so it's appropriate for a login page.
login_redir = RedirectResponse('/login', status_code=303)

def user_auth_before(req, sess):
    # The `auth` key in the request scope is automatically provided
```

```
# to any handler which requests it, and can not be injected
# by the user using query params, cookies, etc, so it should
# be secure to use.
auth = req.scope['auth'] = sess.get('auth', None)
# If the session key is not there, it redirects to the login page.
if not auth: return login_redir
```

Now we pass our `user_auth_before` function as the first argument into a `Beforeware` class. We also pass a list of regular expressions to the `skip` argument, designed to allow users to still get to the home and login pages.

```
beforeware = Beforeware(
    user_auth_before,
    skip=[r'/favicon\.ico', r'/static/.*', r'.*\.\css', r'.*\.\js', '/login', '/']
)

app, rt = fast_app(before=beforeware)
```

Server-sent events (SSE)

With [server-sent events](#), it's possible for a server to send new data to a web page at any time, by pushing messages to the web page. Unlike WebSockets, SSE can only go in one direction: server to client. SSE is also part of the HTTP specification unlike WebSockets which uses its own specification.

FastHTML introduces several tools for working with SSE which are covered in the example below. While concise, there's a lot going on in this function so we've annotated it quite a bit.

```
import random
from asyncio import sleep
from fasthtml.common import *

hdrs=(Script(src="https://unpkg.com/htmx-ext-sse@2.2.1/sse.js"),) ①
app,rt = fast_app(hdrs=hdrs)

@rt
def index():
    return Titled("SSE Random Number Generator",
        P("Generate pairs of random numbers, as the list grows scroll downwards."),
        Div(hx_ext="sse",
            sse_connect="/number-stream",
            hx_swap="beforeend show:bottom",
            sse_swap="message")) ② ③ ④ ⑤

shutdown_event = signal_shutdown() ⑥

async def number_generator():
    while not shutdown_event.is_set():
        data = Article(random.randint(1, 100))
        yield sse_message(data) ⑦ ⑧
        await sleep(1) ⑨
```



```
@rt("/number-stream")
async def get(): return EventStream(number_generator())
```

(10)

- ① Import the HTMX SSE extension
- ② Tell HTMX to load the SSE extension
- ③ Look at the `/number-stream` endpoint for SSE content
- ④ When new items come in from the SSE endpoint, add them at the end of the current content within the div. If they go beyond the screen, scroll downwards
- ⑤ Specify the name of the event. FastHTML's default event name is "message". Only change if you have more than one call to SSE endpoints within a view
- ⑥ Set up the asyncio event loop
- ⑦ Don't forget to make this an `async` function!
- ⑧ Iterate through the asyncio event loop
- ⑨ We yield the data. Data ideally should be comprised of FT components as that plugs nicely into HTMX in the browser
- ⑩ The endpoint view needs to be an async function that returns a [EventStream](#)

Websockets

With websockets we can have bi-directional communications between a browser and client. Websockets are useful for things like chat and certain types of games. While websockets can be used for single direction messages from the server (i.e. telling users that a process is finished), that task is arguably better suited for SSE.

FastHTML provides useful tools for adding websockets to your pages.

```
from fasthtml.common import *
from asyncio import sleep

app, rt = fast_app(ws_hdr=True)

def mk_inp(): return Input(id='msg', autofocus=True)

@rt('/')
async def get(request):
    cts = Div(
        Div(id='notifications'),
        Form(mk_inp(), id='form', ws_send=True),
        hx_ext='ws', ws_connect='/ws')
    return Titled('Websocket Test', cts)

async def on_connect(send):
    print('Connected!')
    await send(Div('Hello, you have connected', id="notifications"))

async def on_disconnect(ws):
    print('Disconnected!')
```

①

②

③

④

⑤

⑥

⑦

```
@app.ws('/ws', conn=on_connect, disconn=on_disconnect)
async def ws(msg:str, send):
    await send(Div('Hello ' + msg, id="notifications"))
    await sleep(2)
    return Div('Goodbye ' + msg, id="notifications"), mk_inp()
```

- ① To use websockets in FastHTML, you must instantiate the app with the `ws_hdr` set to `True`
- ② As we want to use websockets to reset the form, we define the `mk_input` function that can be called from multiple locations
- ③ We create the form and mark it with the `ws_send` attribute, which is documented here in the [HTMX websocket specification](#). This tells HTMX to send a message to the nearest websocket based on the trigger for the form element, which for forms is pressing the `enter` key, an action considered to be a form submission
- ④ This is where the HTMX extension is loaded (`hx_ext='ws'`) and the nearest websocket is defined (`ws_connect='/ws'`)
- ⑤ When a websocket first connects we can optionally have it call a function that accepts a `send` argument. The `send` argument will push a message to the browser.
- ⑥ Here we use the `send` function that was passed into the `on_connect` function to send a `Div` with an `id` of `notifications` that HTMX assigns to the element in the page that already has an `id` of `notifications`
- ⑦ When a websocket disconnects we can call a function which takes no arguments. Typically the role of this function is to notify the server to take an action. In this case, we print a simple message to the console
- ⑧ We use the `app.ws` decorator to mark that `/ws` is the route for our websocket. We also pass in the two optional `conn` and `disconn` parameters to this decorator. As a fun experiment, remove the `conn` and `disconn` arguments and see what happens
- ⑨ Define the `ws` function as `async`. This is necessary for ASGI to be able to serve websockets. The function accepts two arguments, a `msg` that is user input from the browser, and a `send` function for pushing data back to the browser
- ⑩ The `send` function is used here to send HTML back to the page. As the HTML has an `id` of `notifications` , HTMX will overwrite what is already on the page with the same ID
- ⑪ The websocket function can also be used to return a value. In this case, it is a tuple of two HTML elements. HTMX will take the elements and replace them where appropriate. As both have `id` specified (`notifications` and `msg` respectively), they will replace their predecessor on the page.

File Uploads

A common task in web development is uploading files. This example is for uploading files to the hosting server, with information about the uploaded file presented to the user.

File uploads in production can be dangerous

File uploads can be the target of abuse, accidental or intentional. That means users may attempt to upload files that are too large or present a security risk. This is especially of concern for public facing apps. File upload security is outside the scope of this tutorial, for now we suggest reading the [OWASP File Upload Cheat Sheet](#).

```

from fasthtml.common import *
from pathlib import Path

app, rt = fast_app()

upload_dir = Path("filez")
upload_dir.mkdir(exist_ok=True)

@rt('/')
def get():
    return Titled("File Upload Demo",
        Div(cls='grid')(
            Article(
                Form(hx_post="/upload", hx_target="#result-one")(
                    Input(type="file", name="file"),
                    Button("Upload", type="submit", cls='secondary'),
                ),
                Div(id="result-one")
            )
        )
    )


def FileMetaDataCard(file):
    return Article(
        Header(H3(file.filename)),
        Ul(
            Li('Size: ', file.size),
            Li('Content Type: ', file.content_type),
            Li('Headers: ', file.headers),
        )
    )

@rt('/upload')
async def post(file: UploadFile):
    card = FileMetaDataCard(file)
    filebuffer = await file.read()
    (upload_dir / file.filename).write_bytes(filebuffer)
    return card

serve()

```

- ① Every form rendered with the [Form](#) FT component defaults to `enctype="multipart/form-data"`
- ② Don't forget to set the [Input](#) FT Component's type to `file`
- ③ The upload view should receive a [Starlette UploadFile](#) type. You can add other form variables
- ④ We can access the metadata of the card (filename, size, content_type, headers), a quick and safe process
- ⑤ In order to access the contents contained within a file we use the `await` method to `read()` it. As files may be quite large or contain bad data, this is a separate step from accessing metadata
- ⑥ This step shows how to use Python's built-in `pathlib.Path` library to write the file to disk.

 Report an issue



 Tutorials > JS App Walkthrough

JS App Walkthrough

How to build a website with custom JavaScript in FastHTML step-by-step

Installation [↗](#)

You'll need the following software to complete the tutorial, read on for specific installation instructions:

1. Python
2. A Python package manager such as pip (which normally comes with Python) or uv
3. FastHTML
4. Web browser
5. Railway.app account

If you haven't worked with Python before, we recommend getting started with [Miniconda](#).

Note that you will only need to follow the steps in the installation section once per environment. If you create a new repo, you won't need to redo these.

Install FastHTML

For Mac, Windows and Linux, enter:

```
pip install python-fasthtml
```

First steps

By the end of this section you'll have your own FastHTML website with tests deployed to railway.app.

Create a hello world

Create a new folder to organize all the files for your project. Inside this folder, create a file called `main.py` and add the following code to it:

```
main.py

from fasthtml.common import *

app = FastHTML()
rt = app.route

@rt('/')
def get():
```

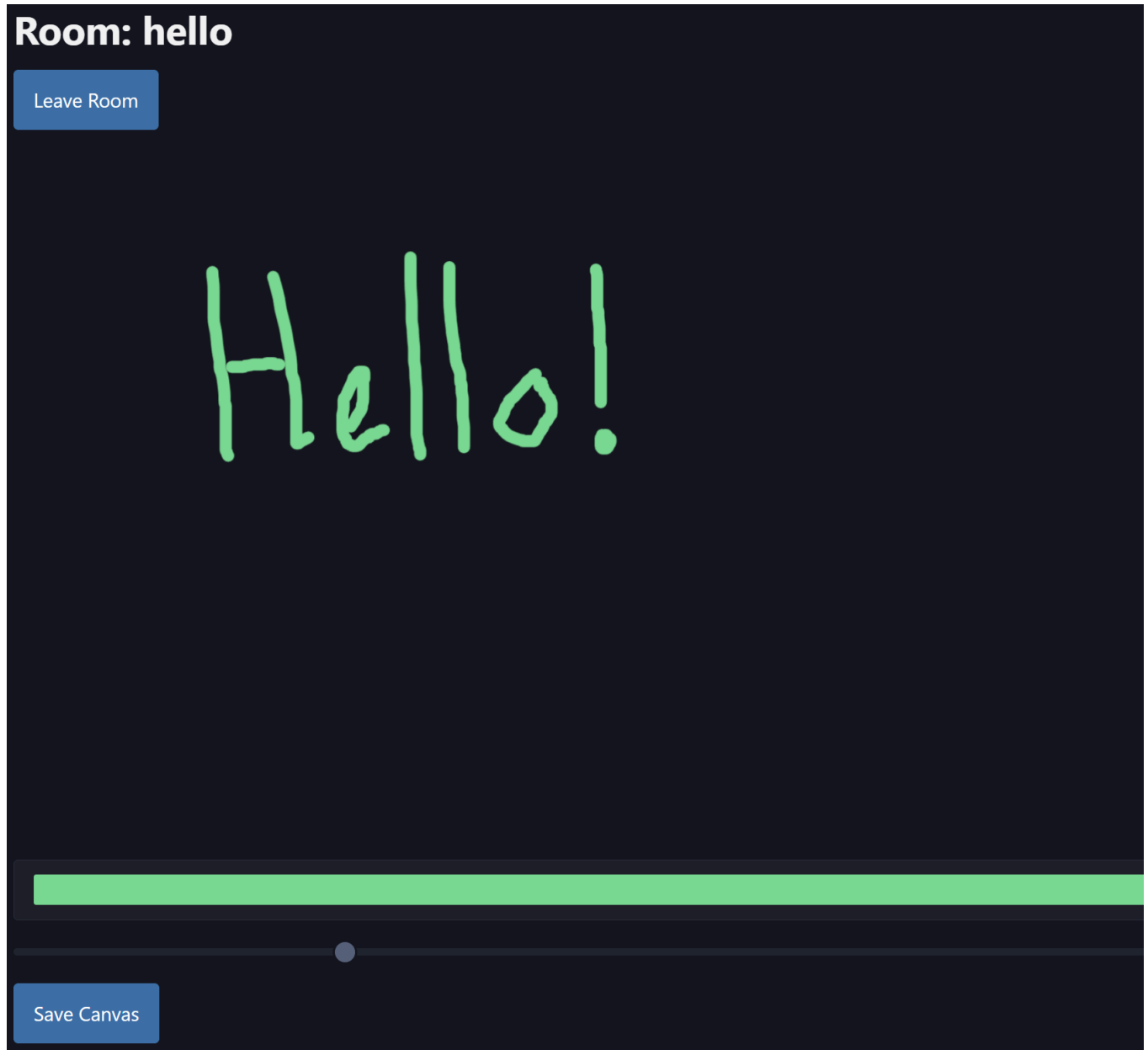
```
return 'Hello, world!'
```

```
serve()
```

Finally, run `python main.py` in your terminal and open your browser to the ‘Link’ that appears.

QuickDraw: A FastHTML Adventure 🎨✨

The end result of this tutorial will be QuickDraw, a real-time collaborative drawing app using FastHTML. Here is what the final site will look like:



QuickDraw

Drawing Rooms

Drawing rooms are the core concept of our application. Each room represents a separate drawing space where a user can let their inner Picasso shine. Here's a detailed breakdown:

1. Room Creation and Storage

```
main.py
```

```

db = database('data/drawapp.db')
rooms = db.t.rooms
if rooms not in db.t:
    rooms.create(id=int, name=str, created_at=str, pk='id')
Room = rooms.dataclass()

@patch
def __ft__(self:Room):
    return Li(A(self.name, href=f"/rooms/{self.id}"))

```

Or you can use our [fast_app](#) function to create a FastHTML app with a SQLite database and dataclass in one line:

```

main.py

def render(room):
    return Li(A(room.name, href=f"/rooms/{room.id}"))

app,rt,rooms,Room = fast_app('data/drawapp.db', render=render, id=int, name=str, creat

```

We are specifying a render function to convert our dataclass into HTML, which is the same as extending the `__ft__` method from the `patch` decorator we used before. We will use this method for the rest of the tutorial since it is a lot cleaner and easier to read.

- We're using a SQLite database (via FastLite) to store our rooms.
- Each room has an id (integer), a name (string), and a created_at timestamp (string).
- The Room dataclass is automatically generated based on this structure.

2. Creating a room

```

main.py

@rt("/")
def get():
    # The 'Input' id defaults to the same as the name, so you can omit it if you wish
    create_room = Form(Input(id="name", name="name", placeholder="New Room Name"),
                        Button("Create Room"),
                        hx_post="/rooms", hx_target="#rooms-list", hx_swap="afterbegin")
    rooms_list = Ul(*rooms(order_by='id DESC'), id='rooms-list')
    return Titled("DrawCollab",
                  H1("DrawCollab"),
                  create_room, rooms_list)

@rt("/rooms")
async def post(room:Room):
    room.created_at = datetime.now().isoformat()
    return rooms.insert(room)

```

- When a user submits the “Create Room” form, this route is called.
- It creates a new Room object, sets the creation time, and inserts it into the database.
- It returns an HTML list item with a link to the new room, which is dynamically added to the room list on the homepage thanks to HTMX.

3. Let's give our rooms shape

main.py

```
@rt("/rooms/{id}")
async def get(id:int):
    room = rooms[id]
    return Titled(f"Room: {room.name}", H1(f"Welcome to {room.name}"), A(Button("Leave
```

- This route renders the interface for a specific room.
- It fetches the room from the database and renders a title, heading, and paragraph.

Here is the full code so far:

main.py

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

def render(room):
    return Li(A(room.name, href=f"/rooms/{room.id}"))

app,rt,rooms,Room = fast_app('data/drawapp.db', render=render, id=int, name=str, creat

@rt("/")
def get():
    create_room = Form(Input(id="name", name="name", placeholder="New Room Name"),
                        Button("Create Room"),
                        hx_post="/rooms", hx_target="#rooms-list", hx_swap="afterbegin")
    rooms_list = Ul(*rooms(order_by='id DESC'), id='rooms-list')
    return Titled("DrawCollab", create_room, rooms_list)

@rt("/rooms")
async def post(room:Room):
    room.created_at = datetime.now().isoformat()
    return rooms.insert(room)

@rt("/rooms/{id}")
async def get(id:int):
    room = rooms[id]
    return Titled(f"Room: {room.name}", H1(f"Welcome to {room.name}"), A(Button("Leave

serve()
```

Now run `python main.py` in your terminal and open your browser to the 'Link' that appears. You should see a page with a form to create a new room and a list of existing rooms.

The Canvas - Let's Get Drawing!

Time to add the actual drawing functionality. We'll use Fabric.js for this:

main.py

```
# ... (keep the previous imports and database setup)

@rt("/rooms/{id}")
async def get(id:int):
```



```

room = rooms[id]
canvas = Canvas(id="canvas", width="800", height="600")
color_picker = Input(type="color", id="color-picker", value="#3CDD8C")
brush_size = Input(type="range", id="brush-size", min="1", max="50", value="10")

js = """
var canvas = new fabric.Canvas('canvas');
canvas.isDrawingMode = true;
canvas.freeDrawingBrush.color = '#3CDD8C';
canvas.freeDrawingBrush.width = 10;

document.getElementById('color-picker').onchange = function() {
    canvas.freeDrawingBrush.color = this.value;
};

document.getElementById('brush-size').oninput = function() {
    canvas.freeDrawingBrush.width = parseInt(this.value, 10);
};
"""

return Titled(f"Room: {room.name}",
              A(Button("Leave Room"), href="/"),
              canvas,
              Div(color_picker, brush_size),
              Script(src="https://cdnjs.cloudflare.com/ajax/libs/fabric.js/5.3.1/fabric.min.js"),
              Script(js))

# ... (keep the serve() part)

```

Now we've got a drawing canvas! FastHTML makes it easy to include external libraries and add custom JavaScript.

Saving and Loading Canvases

Now that we have a working drawing canvas, let's add the ability to save and load drawings. We'll modify our database schema to include a `canvas_data` field, and add new routes for saving and loading canvas data. Here's how we'll update our code:

1. Modify the database schema:

main.py

```
app, rt, rooms, Room = fast_app('data/drawapp.db', render=render, id=int, name=str, creat
```

2. Add a save button that grabs the canvas' state and sends it to the server:

main.py

```

@rt("/rooms/{id}")
async def get(id:int):
    room = rooms[id]
    canvas = Canvas(id="canvas", width="800", height="600")
    color_picker = Input(type="color", id="color-picker", value="#3CDD8C")
    brush_size = Input(type="range", id="brush-size", min="1", max="50", value="10")

```

```
save_button = Button("Save Canvas", id="save-canvas", hx_post=f"/rooms/{id}/save",
# ... (rest of the function remains the same)
```

3. Add routes for saving and loading canvas data:

main.py

```
@rt("/rooms/{id}/save")
async def post(id:int, canvas_data:str):
    rooms.update({'canvas_data': canvas_data}, id)
    return "Canvas saved successfully"

@rt("/rooms/{id}/load")
async def get(id:int):
    room = rooms[id]
    return room.canvas_data if room.canvas_data else "{}"
```

4. Update the JavaScript to load existing canvas data:

main.py

```
js = f"""
    var canvas = new fabric.Canvas('canvas');
    canvas.isDrawingMode = true;
    canvas.freeDrawingBrush.color = '#3CDD8C';
    canvas.freeDrawingBrush.width = 10;
    // Load existing canvas data
    fetch(`/rooms/{id}/load`)
    .then(response => response.json())
    .then(data => {{
        if (data && Object.keys(data).length > 0) {{
            canvas.loadFromJSON(data, canvas.renderAll.bind(canvas));
        }}
    }});

    // ... (rest of the JavaScript remains the same)
    """
```

With these changes, users can now save their drawings and load them when they return to the room. The canvas data is stored as a JSON string in the database, allowing for easy serialization and deserialization. Try it out! Create a new room, make a drawing, save it, and then reload the page. You should see your drawing reappear, ready for further editing.

Here is the completed code:

main.py

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

def render(room):
    return Li(A(room.name, href=f"/rooms/{room.id}"))

app,rt,rooms,Room = fast_app('data/drawapp.db', render=render, id=int, name=str, creat
```

```

@rt("/")
def get():
    create_room = Form(Input(id="name", name="name", placeholder="New Room Name"),
                        Button("Create Room"),
                        hx_post="/rooms", hx_target="#rooms-list", hx_swap="afterbegin")
    rooms_list = Ul(*rooms(order_by='id DESC'), id='rooms-list')
    return Titled("QuickDraw",
                  create_room, rooms_list)

@rt("/rooms")
async def post(room:Room):
    room.created_at = datetime.now().isoformat()
    return rooms.insert(room)

@rt("/rooms/{id}")
async def get(id:int):
    room = rooms[id]
    canvas = Canvas(id="canvas", width="800", height="600")
    color_picker = Input(type="color", id="color-picker", value="#000000")
    brush_size = Input(type="range", id="brush-size", min="1", max="50", value="10")
    save_button = Button("Save Canvas", id="save-canvas", hx_post=f"/rooms/{id}/save",

    js = f"""
    var canvas = new fabric.Canvas('canvas');
    canvas.isDrawingMode = true;
    canvas.freeDrawingBrush.color = '#000000';
    canvas.freeDrawingBrush.width = 10;

    // Load existing canvas data
    fetch(`/rooms/{id}/load`)
    .then(response => response.json())
    .then(data => {{
        if (data && Object.keys(data).length > 0) {{
            canvas.loadFromJSON(data, canvas.renderAll.bind(canvas));
        }}
    }});

    document.getElementById('color-picker').onchange = function() {{
        canvas.freeDrawingBrush.color = this.value;
    }};

    document.getElementById('brush-size').oninput = function() {{
        canvas.freeDrawingBrush.width = parseInt(this.value, 10);
    }};
    """

    return Titled(f"Room: {room.name}",
                  A(Button("Leave Room"), href="/"),
                  canvas,
                  Div(color_picker, brush_size, save_button),
                  Script(src="https://cdn.jsdelivr.net/npm/fabric.js/5.3.1/fabric.min.js"),
                  Script(js))

@rt("/rooms/{id}/save")

```

```
async def post(id:int, canvas_data:str):
    rooms.update({'canvas_data': canvas_data}, id)
    return "Canvas saved successfully"

@rt("/rooms/{id}/load")
async def get(id:int):
    room = rooms[id]
    return room.canvas_data if room.canvas_data else "{}"

serve()
```

Deploying to Railway

You can deploy your website to a number of hosting providers, for this tutorial we'll be using Railway. To get started, make sure you create an [account](#) and install the [Railway CLI](#). Once installed, make sure to run `railway login` to log in to your account.

To make deploying your website as easy as possible, FastHTML comes with a built in CLI tool that will handle most of the deployment process for you. To deploy your website, run the following command in your terminal in the root directory of your project:

```
fh_railway_deploy quickdraw
```

Note

Your app must be located in a `main.py` file for this to work.

Conclusion: You're a FastHTML Artist Now! 🎨🚀


Congratulations! You've just built a sleek, interactive web application using FastHTML. Let's recap what we've learned:

1. FastHTML allows you to create dynamic web apps with minimal code.
2. We used FastHTML's routing system to handle different pages and actions.
3. We integrated with a SQLite database to store room information and canvas data.
4. We utilized Fabric.js to create an interactive drawing canvas.
5. We implemented features like color picking, brush size adjustment, and canvas saving.
6. We used HTMX for seamless, partial page updates without full reloads.
7. We learned how to deploy our FastHTML application to Railway for easy hosting.

You've taken your first steps into the world of FastHTML development. From here, the possibilities are endless! You could enhance the drawing app further by adding features like:

- Implementing different drawing tools (e.g., shapes, text)
- Adding user authentication
- Creating a gallery of saved drawings
- Implementing real-time collaborative drawing using WebSockets

Whatever you choose to build next, FastHTML has got your back. Now go forth and create something awesome! Happy coding! 🎨🚀

 [Report an issue](#)

FT Components

FT components turn Python objects into HTML.

FT, or ‘FastTags’, are the display components of FastHTML. In fact, the word “components” in the context of FastHTML is often synonymous with **FT**.

For example, when we look at a FastHTML app, in particular the views, as well as various functions and other objects, we see something like the code snippet below. It’s the `return` statement that we want to pay attention to:

```
from fasthtml.common import *

def example():
    # The code below is a set of ft components
    return Div(
        H1("FastHTML APP"),
        P("Let's do this"),
        cls="go"
    )
```

Let’s go ahead and call our function and print the result:

```
example()

<div class="go">
  <h1>FastHTML APP</h1>
  <p>Let's do this</p>
</div>
```

As you can see, when returned to the user from a Python callable, like a function, the ft components are transformed into their string representations of XML or XML-like content such as HTML. More concisely, *ft turns Python objects into HTML*.

Now that we know what ft components look and behave like we can begin to understand them. At their most fundamental level, ft components:

1. Are Python callables, specifically functions, classes, methods of classes, lambda functions, and anything else called with parenthesis that returns a value.
2. Return a sequence of values which has three elements:
 1. The tag to be generated
 2. The content of the tag, which is a tuple of strings/tuples. If a tuple, it is the three-element structure of an ft component

3. A dictionary of XML attributes and their values

3. FastHTML's default ft components words begin with an uppercase letter. Examples include `Title()`, `Ul()`, and `Div()`. Custom components have included things like `BlogPost` and `CityMap`.

How FastHTML names ft components

When it comes to naming ft components, FastHTML appears to break from PEP8. Specifically, PEP8 specifies that when naming variables, functions and instantiated classes we use the `snake_case_pattern`. That is to say, lowercase with words separated by underscores. However, FastHTML uses `PascalCase` for ft components.

There's a couple of reasons for this:

1. ft components can be made from any callable type, so adhering to any one pattern doesn't make much sense
2. It makes for easier reading of FastHTML code, as anything that is PascalCase is probably an ft component

Default FT components

FastHTML has over 150 **FT** components designed to accelerate web development. Most of these mirror HTML tags such as `<div>`, `<p>`, `<a>`, `<title>`, and more. However, there are some extra tags added, including:

- `Titled`, a combination of the `Title()` and `H1()` tags
- `Socials`, renders popular social media tags

The `fasthtml.ft` Namespace

Some people prefer to write code using namespaces while adhering to PEP8. If that's a preference, projects can be coded using the `fasthtml.ft` namespace.

```
from fasthtml import ft

ft.Ul(
    ft.Li("one"),
    ft.Li("two"),
    ft.Li("three")
)
```

```
<ul>
  <li>one</li>
  <li>two</li>
  <li>three</li>
</ul>
```

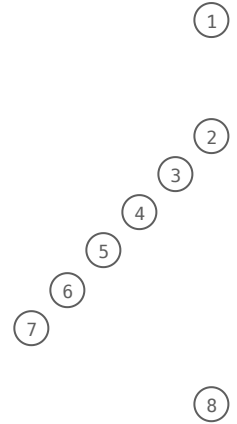
Attributes [↗](#)

This example demonstrates many important things to know about how ft components handle attributes.

```

1  #| echo: False
2  Label(
3      "Choose an option",
4      Select(
5          Option("one", value="1", selected=True),
6          Option("two", value="2", selected=False),
7          Option("three", value=3),
8          cls="selector",
9          _id="counter",
10         **{'@click':"alert('Clicked');"},
11     ),
12     _for="counter",
13 )

```



- ① Line 2 demonstrates that FastHTML appreciates `Label` s surrounding their fields.
- ② On line 5, we can see that attributes set to the `boolean` value of `True` are rendered with just the name of the attribute.
- ③ On line 6, we demonstrate that attributes set to the `boolean` value of `False` do not appear in the rendered output.
- ④ Line 7 is an example of how integers and other non-string values in the rendered output are converted to strings.
- ⑤ Line 8 is where we set the HTML class using the `cls` argument. We use `cls` here as `class` is a reserved word in Python. During the rendering process this will be converted to the word “class”.
- ⑥ Line 9 demonstrates that any named argument passed into an ft component will have the leading underscore stripped away before rendering. Useful for handling reserved words in Python.
- ⑦ On line 10 we have an attribute name that cannot be represented as a python variable. In cases like these, we can use an unpacked `dict` to represent these values.
- ⑧ The use of `_for` on line 12 is another demonstration of an argument having the leading underscore stripped during render. We can also use `fr` as that will be expanded to `for` .

This renders the following HTML snippet:

```

Label(
    "Choose an option",
    Select(
        Option("one", value="1", selected=True),
        Option("two", value="2", selected=False),
        Option("three", value=3), # <4>,
        cls="selector",
        _id="counter",
        **{'@click':"alert('Clicked');"},
    ),
    _for="counter",
)

```

```

<label for="counter">
Choose an option
<select id="counter" @click="alert(&#x27;Clicked&#x27;);" class="selector" name="cc
<option value="1" selected>one</option>

```



```
<option value="2" >two</option>
<option value="3">three</option>
</select>
</label>
```

Defining new ft components


It is possible and sometimes useful to create your own ft components that generate non-standard tags that are not in the FastHTML library. FastHTML supports created and defining those new tags flexibly.

For more information, see the [Defining new ft components](#) reference page.

FT components and type hints

If you use type hints, we strongly suggest that FT components be treated as the `Any` type.

The reason is that FastHTML leverages python's dynamic features to a great degree. Especially when it comes to `FT` components, which can evaluate out to be `FT|str|None|tuple` as well as anything that supports the `__ft__`, `__html__`, and `__str__` method. That's enough of the Python stack that assigning anything but `Any` to be the FT type will prove an exercise in frustration.

 [Report an issue](#)



MiniDataAPI Spec

The **MiniDataAPI** is a persistence API specification that designed to be small and relatively easy to implement across a wide range of datastores. While early implementations have been SQL-based, the specification can be quickly implemented in key/value stores, document databases, and more.

Work in Progress

The MiniData API spec is a work in progress, subject to change. While the majority of design is complete, expect there could be breaking changes.

Why?

The MiniDataAPI specification allows us to use the same API for many different database engines. Any application using the MiniDataAPI spec for interacting with its database requires no modification beyond import and configuration changes to switch database engines. For example, to convert an application from Fastlite running SQLite to FastSQL running PostgreSQL, should require only changing these two lines:

FastLite version

```
from fastlite import *  
db = Database('test.db')
```

FastSQL version

```
from fastsql import *  
db = Database('postgres:...')
```

As both libraries adhere to the MiniDataAPI specification, the rest of the code in the application should remain the same. The advantage of the MiniDataAPI spec is that it allows people to use whatever datastores they have access to or prefer.

Note

Switching databases won't migrate any existing data between databases.

Easy to learn, quick to implement

The MiniDataAPI specification is designed to be easy-to-learn and quick to implement. It focuses on straightforward Create, Read, Update, and Delete (CRUD) operations.

MiniDataAPI databases aren't limited to just row-based systems. In fact, the specification is closer in design to a key/value store than a set of records. What's exciting about this is we can write implementations for tools like Python dict stored as JSON, Redis, and even the venerable ZODB.

Limitations of the MiniDataAPI Specification

“Mini refers to the lightweightness of specification, not the data.”

– Jeremy Howard

The advantages of the MiniDataAPI come at a cost. The MiniDataAPI specification focuses a very small set of features compared to what can be found in full-fledged ORMs and query languages. It intentionally avoids nuances or sophisticated features.

This means the specification does not include joins or formal foreign keys. Complex data stored over multiple tables that require joins isn't handled well. For this kind of scenario it's probably for the best to use more sophisticated ORMs or even direct database queries.

Summary of the MiniDataAPI Design

- Easy-to-learn
- Relative quick to implement for new database engines
- An API for CRUD operations
- For many different types of databases including row- and key/value-based designs
- Intentionally small in terms of features: no joins, no foreign keys, no database specific features
- Best for simpler designs, complex architectures will need more sophisticated tools.

Connect/construct the database

We connect or construct the database by passing in a string connecting to the database endpoint or a filepath representing the database's location. While this example is for SQLite running in memory, other databases such as PostgreSQL, Redis, MongoDB, might instead use a URI pointing at the database's filepath or endpoint. The method of connecting to a DB is *not* part of this API, but part of the underlying library. For instance, for fastlite:

```
db = database(':memory:')
```

Here's a complete list of the available methods in the API, all documented below (assuming **db** is a database and **t** is a table):

- **db.create**
- **t.insert**
- **t.delete**
- **t.update**
- **t[key]**
- **t(...)**
- **t.xtra**

Tables

For the sake of expediency, this document uses a SQL example. However, tables can represent anything, not just the fundamental construct of a SQL databases. They might represent keys within a key/value structure or files on a hard-drive.

Creating tables

We use a `create()` method attached to `Database` object (`db` in our example) to create the tables.

```
class User: name:str; email: str; year_started:int
users = db.create(User, pk='name')
users
```

```
<Table user (name, email, year_started)>
```

```
class User: name:str; email: str; year_started:int
users = db.create(User, pk='name')
users
```

```
<Table user (name, email, year_started)>
```

If no `pk` is provided, `id` is assumed to be the primary key. Regardless of whether you mark a class as a dataclass or not, it will be turned into one – specifically into a [flexiclass](#).

```
@dataclass
class Todo: id: int; title: str; detail: str; status: str; name: str
todos = db.create(Todo)
todos
```

```
<Table todo (id, title, detail, status, name)>
```

Compound primary keys

The MiniData API spec supports compound primary keys, where more than one column is used to identify records. We'll also use this example to demonstrate creating a table using a dict of keyword arguments.

```
class Publication: authors: str; year: int; title: str
publications = db.create(Publication, pk=('authors', 'year'))
```

Transforming tables

Depending on the database type, this method can include transforms - the ability to modify the tables. Let's go ahead and add a password field for our table called `pwd`.

```
class User: name:str; email: str; year_started:int; pwd:str
users = db.create(User, pk='name', transform=True)
users
```

```
<Table user (name, email, year_started, pwd)>
```

Manipulating data

The specification is designed to provide as straightforward CRUD API (Create, Read, Update, and Delete) as possible. Additional features like joins are out of scope.

`.insert()`

Add a new record to the database. We want to support as many types as possible, for now we have tests for Python classes, dataclasses, and dicts. Returns an instance of the new record.

Here's how to add a record using a Python class:

```
users.insert(User(name='Braden', email='b@example.com', year_started=2018))
User(name='Braden', email='b@example.com', year_started=2018, pwd=None)
```

We can also use keyword arguments directly:

```
users.insert(name='Alma', email='a@example.com', year_started=2019)
User(name='Alma', email='a@example.com', year_started=2019, pwd=None)
```

And now Charlie gets added via a Python dict.

```
users.insert({'name': 'Charlie', 'email': 'c@example.com', 'year_started': 2018})
User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)
```

And now TODOs. Note that the inserted row is returned:

```
todos.insert(Todo(title='Write MiniDataAPI spec', status='open', name='Braden'))
todos.insert(title='Implement SSE in FastHTML', status='open', name='Alma')
todo = todos.insert(dict(title='Finish development of FastHTML', status='closed', name='Charlie'))
Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed', name='Charlie')
```

Let's do the same with the **Publications** table.

```
publications.insert(publication(publication=Publication(authors='Alma', year=2019, title='FastHTML')))
publications.insert(authors='Alma', year=2030, title='FastHTML and beyond')
publication= publications.insert((dict(authors='Alma', year=2035, title='FastHTML, the early years')))
Publication(authors='Alma', year=2035, title='FastHTML, the early years')
```

Square bracket search []

Get a single record by entering a primary key into a table object within square brackets. Let's see if we can find Alma.

```
user = users['Alma']
User(name='Alma', email='a@example.com', year_started=2019, pwd=None)
```

If no record is found, a **NotFoundError** error is raised. Here we look for David, who hasn't yet been added to our users table.

```
try: users['David']
except NotFoundError: print(f'User not found')
User not found
```

Here's a demonstration of a ticket search, demonstrating how this works with non-string primary keys.

```
todos[1]
```

```
Todo(id=1, title='Write MiniDataAPI spec', detail=None, status='open',  
name='Braden')
```

Compound primary keys can be supplied in lists or tuples, in the order they were defined. In this case it is the **authors** and **year** columns.

Here's a query by compound primary key done with a **list**:

```
publications[['Alma', 2019]]
```

```
Publication(authors='Alma', year=2019, title='FastHTML')
```

Here's the same query done directly with index args.

```
publications['Alma', 2030]
```

```
Publication(authors='Alma', year=2030, title='FastHTML and beyond')
```

Parentheses search ()

Get zero to many records by entering values with parentheses searches. If nothing is in the parentheses, then everything is returned.

```
users()
```

```
[User(name='Braden', email='b@example.com', year_started=2018, pwd=None),  
User(name='Alma', email='a@example.com', year_started=2019, pwd=None),  
User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)]
```

We can order the results.

```
users(order_by='name')
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None),  
User(name='Braden', email='b@example.com', year_started=2018, pwd=None),  
User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)]
```

We can filter on the results:

```
users(where="name='Alma'")
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None)]
```

Generally you probably want to use placeholders, to avoid SQL injection attacks:

```
users("name=?", ('Alma',))
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None)]
```

We can limit results with the **limit** keyword:

```
users(limit=1)
```

```
[User(name='Braden', email='b@example.com', year_started=2018, pwd=None)]
```

If we're using the `limit` keyword, we can also use the `offset` keyword to start the query later.

```
users(limit=5, offset=1)
```

```
[User(name='Alma', email='a@example.com', year_started=2019, pwd=None),  
 User(name='Charlie', email='c@example.com', year_started=2018, pwd=None)]
```

.update()

Update an existing record of the database. Must accept Python dict, dataclasses, and standard classes. Uses the primary key for identifying the record to be changed. Returns an instance of the updated record.

Here's with a normal Python class:

```
user
```

```
User(name='Alma', email='a@example.com', year_started=2019, pwd=None)
```

```
user.year_started = 2099  
users.update(user)
```

```
User(name='Alma', email='a@example.com', year_started=2099, pwd=None)
```

Or use a dict:

```
users.update(dict(name='Alma', year_started=2199, email='a@example.com'))
```

```
User(name='Alma', email='a@example.com', year_started=2199, pwd=None)
```

Or use kwargs:

```
users.update(name='Alma', year_started=2149)
```

```
User(name='Alma', email='a@example.com', year_started=2149, pwd=None)
```

If the primary key doesn't match a record, raise a `NotFoundError`.

John hasn't started with us yet so doesn't get the chance yet to travel in time.

```
try: users.update(User(name='John', year_started=2024, email='j@example.com'))  
except NotFoundError: print('User not found')
```

```
User not found
```

.delete()

Delete a record of the database. Uses the primary key for identifying the record to be removed. Returns a table object.

Charlie decides to not travel in time. He exits our little group.

```
users.delete('Charlie')
```

```
<Table user (name, email, year_started, pwd)>
```

If the primary key value can't be found, raises a `NotFoundError`.

```
try: users.delete('Charlies')
except NotFoundError: print('User not found')
```

```
User not found
```

In John's case, he isn't time travelling with us yet so can't be removed.

```
try: users.delete('John')
except NotFoundError: print('User not found')
```

```
User not found
```

Deleting records with compound primary keys requires providing the entire key.

```
publications.delete(['Alma' , 2035])
```

```
<Table publication (authors, year, title)>
```

in keyword

Are **Alma** and **John** contained **in** the Users table? Or, to be technically precise, is the item with the specified primary key value **in** this table?

```
'Alma' in users, 'John' in users
```

```
(True, False)
```

Also works with compound primary keys, as shown below. You'll note that the operation can be done with either a **list** or **tuple**.

```
['Alma', 2019] in publications
```

```
True
```

And now for a **False** result, where John has no publications.

```
('John', 1967) in publications
```

```
False
```

.extra()

If we set fields within the **.extra** function to a particular value, then indexing is also filtered by those. This applies to every database method except for record creation. This makes it easier to limit users (or other objects) access to only things for which they have permission.

For example, if we query all our records below without setting values via the **.extra** function, we can see todos for everyone. Pay special attention to the **id** values of all three records, as we are about to filter most of them away.

```
todos()
```

```
[Todo(id=1, title='Write MiniDataAPI spec', detail=None, status='open',
name='Braden'),
 Todo(id=2, title='Implement SSE in FastHTML', detail=None, status='open',
```



```
name='Alma'),
  Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')]
```

Let's use `.extra` to constrain results just to Charlie. We set the `name` field in Todos, but it could be any field defined for this table.

```
todos.extra(name='Charlie')
```

We've now set a field to a value with `.extra`, if we loop over all the records again, only those assigned to records with a `name` of `Charlie` will be displayed.

```
todos()
[Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')]
```

The `in` keyword is also affected. Only records with a `name` of Charlie will evaluate to be `True`. Let's demonstrate by testing it with a Charlie record:

```
ct = todos[3]
ct
Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')
```

Charlie's record has an ID of 3. Here we demonstrate that Charlie's TODO can be found in the list of todos:

```
ct.id in todos
```

```
True
```

If we try `in` with the other IDs the query fails because the filtering is now set to just records with a name of Charlie.

```
1 in todos, 2 in todos
```

```
(False, False)
```

```
try: todos[2]
except NotFoundError: print('Record not found')
```

```
Record not found
```

We are also constrained by what records we can update. In the following example we try to update a TODO not named 'Charlie'. Because the name is wrong, the `.update` function will raise a `NotFoundError`.

```
try: todos.update(Todo(id=1, title='Finish MiniDataAPI Spec', status='closed', name='B
except NotFoundError as e: print('Record not updated')
```

```
Record not updated
```

Unlike poor Braden, Charlie isn't filtered out. Let's update his TODO.

```
todos.update(Todo(id=3, title='Finish development of FastHTML', detail=None, status='c
```

```
Todo(id=3, title='Finish development of FastHTML', detail=None, status='closed',
name='Charlie')
```

Finally, once constrained by `.xtra`, only records with Charlie as the name can be deleted.

```
try: todos.delete(1)
except NotFoundError as e: print('Record not updated')
```

```
Record not updated
```

Charlie's TODO was to finish development of FastHTML. While the framework will stabilize, like any good project it will see new features added and the odd bug corrected for many years to come. Therefore, Charlie's TODO is nonsensical. Let's delete it.

```
todos.delete(ct.id)
```

```
<Table todo (id, title, detail, status, name)>
```

When a TODO is inserted, the `xtra` fields are automatically set. This ensures that we don't accidentally, for instance, insert items for others users. Note that here we don't set the `name` field, but it's still included in the resultant row:

```
ct = todos.insert(Todo(title='Rewrite personal site in FastHTML', status='open'))
ct
```

```
Todo(id=3, title='Rewrite personal site in FastHTML', detail=None, status='open',
name='Charlie')
```

If we try to change the username to someone else, the change is ignored, due to `xtra`:

```
ct.name = 'Braden'
todos.update(ct)
```

```
Todo(id=3, title='Rewrite personal site in FastHTML', detail=None, status='open',
name='Charlie')
```

SQL-first design

```
users = None
User = None
```

```
users = db.t.user
users
```

```
<Table user (name, email, year_started, pwd)>
```

(This section needs to be documented properly.)

From the table objects we can extract a Dataclass version of our tables. Usually this is given an singular uppercase version of our table name, which in this case is `User`.

```
User = users.dataclass()
```

```
User(name='Braden', email='b@example.com', year_started=2018)
```

```
User(name='Braden', email='b@example.com', year_started=2018, pwd=UNSET)
```

Implementations

Implementing MiniDataAPI for a new datastore

For creating new implementations, the code examples in this specification are the test case for the API. New implementations should pass the tests in order to be compliant with the specification.

Implementations

- [fastlite](#) - The original implementation, only for Sqlite
- [fastsql](#) - An SQL database agnostic implementation based on the excellent SQLAlchemy library.

 [Report an issue](#)



OAuth

OAuth is an open standard for ‘access delegation’, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords. It is the mechanism that enables “Log in with Google” on many sites, saving you from having to remember and manage yet another password. Like many auth-related topics, there’s a lot of depth and complexity to the OAuth standard, but once you understand the basic usage it can be a very convenient alternative to managing your own user accounts.

On this page you’ll see how to use OAuth with FastHTML to implement some common pieces of functionality.

In FastHTML you set up a client like [GoogleAppClient](#). The client is responsible for storing the client ID and client secret, and for handling the OAuth flow. Let’s run through three examples, illustrating some important concepts across three different OAuth providers.

A Minimal Login Flow (GitHub)

Let’s begin by building a minimal ‘Sign in with GitHub’ flow. This will demonstrate the basic steps of OAuth.

OAuth requires a “provider” (in this case, GitHub) to authenticate the user. So the first step when setting up our app is to register with GitHub to set things up.

Go to <https://github.com/settings/developers> and click “New OAuth App”. Fill in the form with the following values, then click ‘Register application’.

- Application name: Your app name
- Homepage URL: <http://localhost:8000> (or whatever URL you’re using - you can change this later)
- Authorization callback URL: http://localhost:8000/auth_redirect (you can modify this later too)

New OAuth Application

github.com/settings/applications/new

Settings / Developer Settings

Register a new OAuth application

Application name *

Something users will recognize and trust.

Homepage URL *

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

Your application's callback URL. Read our [OAuth documentation](#) for more information.

☐ **Enable Device Flow**

Allow this OAuth App to authorize users via the Device Flow.
Read the [Device Flow documentation](#) for more information.

Register application **Cancel**

[Terms](#) [Privacy](#) [Security](#) [Status](#) [Docs](#) [Contact](#) [Manage cookies](#) [Do not share my personal information](#)

© 2024 GitHub, Inc.

After you register, you'll see a screen where you can view the client ID and generate a client secret. Store these values in a safe place. You'll use them to create a [GitHubAppClient](#) object in FastHTML.

This `client` object is responsible for handling the parts of the OAuth flow which depend on direct communication between your app and GitHub, as opposed to interactions which go through the user's browser via redirects.

Here is how to setup the client object:

```
client = GitHubAppClient(
    client_id="your_client_id",
    client_secret="your_client_secret"
)
```

You should also save the path component of the authorization callback URL which you provided on registration.

This route is where GitHub will redirect the user's browser in order to send an authorization code to your app. You should save only the URL's path component rather than the entire URL because you want your code to

work automatically in deployment, when the host and port part of the URL change from `localhost:8000` to your real DNS name.

Save the special authorization callback path under an obvious name:

```
auth_callback_path = "/auth_redirect"
```

Note

It's recommended to store the client ID, and secret, in environment variables, rather than hardcoding them in your code.

When the user visit a normal page of your app, if they are not already logged in, then you'll want to redirect them to your app's login page, which will live at the `/login` path. We accomplish that by using this piece of "beforeware", which defines logic which runs before other work for all routes except ones we specify to be skipped:

```
def before(req, session):
    auth = req.scope['auth'] = session.get('user_id', None)
    if not auth: return RedirectResponse('/login', status_code=303)
    counts.xtra(name=auth)
    bware = Beforeware(before, skip=['/login', auth_callback_path])
```

We configure the beforeware to skip `/login` because that's where the user goes to login, and we also skip the special authorization callback path because that is used by OAuth itself to receive information from GitHub.

It's only at your login page that we start the OAuth flow. To start the OAuth flow, you need to give the user a link to GitHub's login for your app. You'll need the `client` object to generate that link, and the client object will in turn need the full authorization callback URL, which we need to build from the authorization callback path, so it is a multi-step process to produce this GitHub login link.

Here is an implementation of your own `/login` route handler. It generates the GitHub login link and presents it to the user:

```
@app.get('/login')
def login(request):
    redir = redir_url(request, auth_callback_path)
    login_link = client.login_link(redir)
    return P(A('Login with GitHub', href=login_link))
```

Once the user follows that link, GitHub will ask them to grant permission to your app to access their GitHub account. If they agree, GitHub will redirect them back to your app's authorization callback URL, carrying an authorization code which your app can use to generate an access token. To receive this code, you need to set up a route in FastHTML that listens for requests at the authorization callback path. For example:

```
@app.get(auth_callback_path)
def auth_redirect(code:str):
    return P(f"code: {code}")
```

This authorization code is temporary, and is used by your app to directly ask the provider for user information like an access token.

To recap, you can think of the exchange so far as:

- User to us: “I want to log in with you, app.”
- Us to User: “Okay but first, here’s a special link to log in with GitHub”
- User to GitHub: “I want to log in with you, GitHub, to use this app.”
- GitHub to User: “OK, redirecting you back to the app’s URL (with an auth code)”
- User to Us: “Hi again, app. Here’s the GitHub auth code you need to ask GitHub for info about me”
(delivered via `/auth_redirect?code=...`)

The final steps we need to implement are as follows:

- Us to GitHub: “A user just gave me this auth code. May I have the user info (e.g., an access token)?”
- GitHub to us: “Since you have an auth code, here’s the user info”

It’s critical for us to derive the user info from the auth code immediately in the authorization callback, because the auth code may be used only once. So we use it that once in order to get information like an access token, which will remain valid for longer.

To go from the auth code to user info, you use `info = client.retr_info(code, redirect_uri)`. From the user info, you can extract the `user_id`, which is a unique identifier for the user:

```
@app.get(auth_callback_path)
def auth_redirect(code:str, request):
    redir = redir_url(request, auth_callback_path)
    user_info = client.retr_info(code, redir)
    user_id = info[client.id_key]
    return P(f"User id: {user_id}")
```

But we want the user ID not to print it but to remember the user.

So let us store it in the `session` object, to remember who is logged in:

```
@app.get(auth_callback_path)
def auth_redirect(code:str, request, session):
    redir = redir_url(request, auth_callback_path)
    user_info = client.retr_info(code, redir)
    user_id = user_info[client.id_key] # get their ID
    session['user_id'] = user_id # save ID in the session
    return RedirectResponse('/', status_code=303)
```

The session object is derived from values visible to the user’s browser, but it is cryptographically signed so the user can’t read it themselves. This makes it safe to store even information we don’t want to expose to the user.

For larger quantities of data, we’d want to save that information in a database and use the session to hold keys to lookup information from that database.

Here’s a minimal app that puts all these pieces together. It uses the user info to get the `user_id`. It stores that in the session object. It then uses the `user_id` as a key into a database, which tracks how frequently every user has hit an increment button.

```

import os
from fasthtml.common import *
from fasthtml.oauth import GitHubAppClient, redirect_url

db = database('data/counts.db')
counts = db.t.counts
if counts not in db.t: counts.create(dict(name=str, count=int), pk='name')
Count = counts.dataclass()

# Auth client setup for GitHub
client = GitHubAppClient(os.getenv("AUTH_CLIENT_ID"),
                        os.getenv("AUTH_CLIENT_SECRET"))
auth_callback_path = "/auth_redirect"

def before(req, session):
    # if not logged in, we send them to our login page
    # logged in means:
    # - 'user_id' in the session object,
    # - 'auth' in the request object
    auth = req.scope['auth'] = session.get('user_id', None)
    if not auth: return RedirectResponse('/login', status_code=303)
    counts.xtra(name=auth)
bware = Beforeware(before, skip=['/login', auth_callback_path])

app = FastHTML(before=bware)

# User asks us to Login
@app.get('/login')
def login(request):
    redirect = redirect_url(request, auth_callback_path)
    login_link = client.login_link(redirect)
    # we tell user to login at github
    return P(A('Login with GitHub', href=login_link))

# User comes back to us with an auth code from Github
@app.get(auth_callback_path)
def auth_redirect(code:str, request, session):
    redirect = redirect_url(request, auth_callback_path)
    user_info = client.retr_info(code, redirect)
    user_id = user_info[client.id_key] # get their ID
    session['user_id'] = user_id # save ID in the session
    # create a db entry for the user
    if user_id not in counts: counts.insert(name=user_id, count=0)
    return RedirectResponse('/', status_code=303)

@app.get('/')
def home(auth):
    return Div(
        P("Count demo"),
        P(f"Count: ", Span(counts[auth].count, id='count')),
        Button('Increment', hx_get='/increment', hx_target='#count'),
        P(A('Logout', href='/logout'))
    )

```



```

@app.get('/increment')
def increment(auth):
    c = counts[auth]
    c.count += 1
    return counts.upsert(c).count

@app.get('/logout')
def logout(session):
    session.pop('user_id', None)
    return RedirectResponse('/login', status_code=303)

serve()

```

Some things to note:

- The `before` function is used to check if the user is authenticated. If not, they are redirected to the login page.
- To log the user out, we remove the user ID from the session.
- Calling `counts.xtra(name=auth)` ensures that only the row corresponding to the current user is accessible when responding to a request. This is often nicer than trying to remember to filter the data in every route, and lowers the risk of accidentally leaking data.
- In the `auth_redirect` route, we store the user ID in the session and create a new row in the `user_counts` table if it doesn't already exist.

You can find more heavily-commented version of this code in the [oauth directory in fasthtml-example](#), along with an even more minimal example. More examples may be added in the future.

Revoking Tokens (Google)

When the user in the example above logs out, we remove their user ID from the session. However, the user is still logged in to GitHub. If they click 'Login with GitHub' again, they'll be redirected back to our site without having to log in again. This is because GitHub remembers that they've already granted our app permission to access their account. Most of the time this is convenient, but for testing or security purposes you may want a way to revoke this permission.

As a user, you can usually revoke access to an app from the provider's website (for example, <https://github.com/settings/applications>). But as a developer, you can also revoke access programmatically - at least with some providers. This requires keeping track of the access token (stored in `client.token["access_token"]` after you call `retr_info`), and sending a request to the provider's revoke URL:

```

auth_revoke_url = "https://accounts.google.com/o/oauth2/revoke"
def revoke_token(token):
    response = requests.post(auth_revoke_url, params={"token": token})
    return response.status_code == 200 # True if successful

```

Not all providers support token revocation, and it is not built into FastHTML clients at the moment.

Using State (Hugging Face)

Imagine a user (not logged in) comes to your AI image editing site, starts testing things out, and then realizes they need to sign in before they can click “Run (Pro)” on the edit they’re working on. They click “Sign in with Hugging Face”, log in, and are redirected back to your site. But now they’ve lost their in-progress edit and are left just looking at the homepage! This is an example of a case where you might want to keep track of some additional state. Another strong use case for being able to pass some unique state through the OAuth flow is to prevent something called a [CSRF attack](#). To add a state string to the OAuth flow, you can use `client.login_link_with_state(state)` instead of `client.login_link()`, like so:

```
# in login page:
link = A('Login with GitHub', href=client.login_link_with_state(state='current_prompt:

# in auth_redirect:
@app.get('/auth_redirect')
def auth_redirect(code:str, session, state:str=None):
    print(f"state: {state}") # Use as needed
    ...
```

The state string is passed through the OAuth flow and back to your site.

A Work in Progress

This page (and OAuth support in FastHTML) is a work in progress. Questions, PRs, and feedback are welcome!

 [Report an issue](#)



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 flexibility 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 wildy that putting all the routes into `main.py` becomes unwieldy. 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")),
        Hr(),
        P(A(link=uri("books:list"))("Books")),
    )

serve()
```

②

③

- 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](#)



WebSockets

Websockets are a protocol for two-way, persistent communication between a client and server. This is different from HTTP, which uses a request/response model where the client sends a request and the server responds. With websockets, either party can send messages at any time, and the other party can respond.

This allows for different applications to be built, including things like chat apps, live-updating dashboards, and real-time collaborative tools, which would require constant polling of the server for updates with HTTP.

In FastHTML, you can create a websocket route using the `@app.ws` decorator. This decorator takes a route path, and optional `conn` and `disconn` parameters representing the `on_connect` and `on_disconnect` callbacks in websockets, respectively. The function decorated by `@app.ws` is the main function that is called when a message is received.

Here's an example of a basic websocket route:

```
@app.ws('/ws', conn=on_conn, disconn=on_disconn)
async def on_message(msg:str, send):
    await send(Div('Hello ' + msg, id='notifications'))
    await send(Div('Goodbye ' + msg, id='notifications'))
```

The `on_message` function is the main function that is called when a message is received and can be named however you like. Similar to standard routes, the arguments to `on_message` are automatically parsed from the websocket payload for you, so you don't need to manually parse the message content. However, certain argument names are reserved for special purposes. Here are the most important ones:

- `send` is a function that can be used to send text data to the client.
- `data` is a dictionary containing the data sent by the client.
- `ws` is a reference to the websocket object.

For example, we can send a message to the client that just connected like this:

```
async def on_conn(send):
    await send(Div('Hello, world!'))
```

Or if we receive a message from the client, we can send a message back to them:

```
@app.ws('/ws', conn=on_conn, disconn=on_disconn)
async def on_message(msg:str, send):
    await send(Div('You said: ' + msg, id='notifications'))
    # or...
    return Div('You said: ' + msg, id='notifications')
```

On the client side, we can use HTMX's websocket extension to open a websocket connection and send/receive messages. For example:

```
from fasthtml.common import *

app = FastHTML(ws_hdr=True)

@app.get('/')
def home():
    cts = Div(
        Div(id='notifications'),
        Form(Input(id='msg'), id='form', ws_send=True),
        hx_ext='ws', ws_connect='/ws')
    return Titled('Websocket Test', cts)
```

This will create a websocket connection to the server on route `/ws`, and send any form submissions to the server via the websocket. The server will then respond by sending a message back to the client. The client will then update the message div with the message from the server using Out of Band Swaps, which means that the content is swapped with the same id without reloading the page.

Note

Make sure you set `ws_hdr=True` when creating your [FastHTML](#) object if you want to use websockets so the extension is loaded.

Putting it all together, the code for the client and server should look like this:

```
from fasthtml.common import *

app = FastHTML(ws_hdr=True)
rt = app.route

@rt('/')
def get():
    cts = Div(
        Div(id='notifications'),
        Form(Input(id='msg'), id='form', ws_send=True),
        hx_ext='ws', ws_connect='/ws')
    return Titled('Websocket Test', cts)

@app.ws('/ws')
async def ws(msg:str, send):
    await send(Div('Hello ' + msg, id='notifications'))

serve()
```

This is a fairly simple example and could be done just as easily with standard HTTP requests, but it illustrates the basic idea of how websockets work. Let's look at a more complex example next.

Real-Time Chat App

Let's put our new websocket knowledge to use by building a simple chat app. We will create a chat app where multiple users can send and receive messages in real time.

Let's start by defining the app and the home page:

```
from fasthtml.common import *

app = FastHTML(ws_hdr=True)
rt = app.route

msgs = []
@rt('/')
def home(): return Div(
    Div(UL(*[Li(m) for m in msgs], id='msg-list')),
    Form(Input(id='msg'), id='form', ws_send=True),
    hx_ext='ws', ws_connect='/ws')
```

Now, let's handle the websocket connection. We'll add a new route for this along with an `on_conn` and `on_disconn` function to keep track of the users currently connected to the websocket. Finally, we will handle the logic for sending messages to all connected users.

```
users = {}
def on_conn(ws, send): users[str(id(ws))] = send
def on_disconn(ws): users.pop(str(id(ws)), None)


@app.ws('/ws', conn=on_conn, disconn=on_disconn)
async def ws(msg:str):
    msgs.append(msg)
    # Use associated `send` function to send message to each user
    for u in users.values(): await u(UL(*[Li(m) for m in msgs], id='msg-list'))

serve()
```

We can now run this app with `python chat_ws.py` and open multiple browser tabs to `http://localhost:5001`. You should be able to send messages in one tab and see them appear in the other tabs.

A Work in Progress

This page (and Websocket support in FastHTML) is a work in progress. Questions, PRs, and feedback are welcome!

 Report an issue



Custom Components

The majority of the time the default [ft components](#) are all you need (for example `Div`, `P`, `H1`, etc.).

Pre-requisite Knowledge

If you don't know what an ft component is, you should read [the explaining ft components explainer first](#).

However, there are many situations where you need a custom ft component that creates a unique HTML tag (for example `<zero-md></zero-md>`). There are many options in FastHTML to do this, and this section will walk through them. Generally you want to use the highest level option that fits your needs.

Real-world example

[This external tutorial](#) walks through a practical situation where you may want to create a custom HTML tag using a custom ft component. Seeing a real-world example is a good way to understand why the contents of this guide is useful.

NotStr

The first way is to use the `NotStr` class to use an HTML tag as a string. It works as a one-off but quickly becomes harder to work with as complexity grows. However we can see that you can generate the same xml using `NotStr` as the out-of-the-box components.

```
from fasthtml.common import NotStr, Div, to_xml
```

```
div_NotStr = NotStr('<div></div>')  
print(div_NotStr)
```

```
<div></div>
```

Automatic Creation

The next (and better) approach is to let FastHTML generate the component function for you. As you can see in our `assert` this creates a function that creates the HTML just as we wanted. This works even though there is not a `Some_never_before_used_tag` function in the `fasthtml.components` source code (you can verify this yourself by looking at the source code).

Tip

Typically these tags are needed because a CSS or Javascript library created a new XML tag that isn't default HTML. For example the `zero-md` javascript library looks for a `<zero-md></zero-md>` tag to know what to run its javascript code

on. Most CSS libraries work by creating styling based on the `class` attribute, but they can also apply styling to an arbitrary HTML tag that they made up.

```
from fasthtml.components import Some_never_before_used_tag
```

```
Some_never_before_used_tag()
```

```
<some-never-before-used-tag></some-never-before-used-tag>
```

Manual Creation

The automatic creation isn't magic. It's just calling a python function `__getattr__` and you can call it yourself to get the same result.

```
import fasthtml

auto_called = fasthtml.components.Some_never_before_used_tag()
manual_called = fasthtml.components.__getattr__('Some_never_before_used_tag')()

# Proving they generate the same xml
assert to_xml(auto_called) == to_xml(manual_called)
```

Knowing that, we know that it's possible to create a different function that has different behavior than FastHTMLs default behavior by modifying how the `__getattr__` function creates the components! It's only a few lines of code and reading that what it does is a great way to understand components more deeply.

Tip

Dunder methods and functions are special functions that have double underscores at the beginning and end of their name. They are called at specific times in python so you can use them to cause customized behavior that makes sense for your specific use case. They can appear magical if you don't know how python works, but they are extremely commonly used to modify python's default behavior (`__init__` is probably the most common one).

In a module `__getattr__` is called to get an attribute. In `fasthtml.components`, this is defined to create components automatically for you.

For example if you want a component that creates `<path></path>` that doesn't conflict names with `pathlib.Path` you can do that. FastHTML automatically creates new components with a 1:1 mapping and a consistent name, which is almost always what you want. But in some cases you may want to customize that and you can use the `ft_hx` function to do that differently than the default.

```
from fasthtml.common import ft_hx

def ft_path(*c, target_id=None, **kwargs):
    return ft_hx('path', *c, target_id=target_id, **kwargs)

ft_path()
```

```
<path></path>
```

We can add any behavior in that function that we need to, so let's go through some progressively complex examples that you may need in some of your projects.

Underscores in tags

Now that we understand how FastHTML generates components, we can create our own in all kinds of ways. For example, maybe we need a weird HTML tag that uses underscores. FastHTML replaces `_` with `-` in tags because underscores in tags are highly unusual and rarely what you want, though it does come up rarely.

```
def tag_with_underscores(*c, target_id=None, **kwargs):
    return ft_hx('tag_with_underscores', *c, target_id=target_id, **kwargs)

tag_with_underscores()
```

```
<tag_with_underscores></tag_with_underscores>
```

Symbols (ie @) in tags

Sometimes you may need to use a tag that uses characters that are not allowed in function names in python (again, very unusual).

```
def tag_with_AtSymbol(*c, target_id=None, **kwargs):
    return ft_hx('tag-with-@symbol', *c, target_id=target_id, **kwargs)

tag_with_AtSymbol()
```


```
<tag-with-@symbol></tag-with-@symbol>
```

Symbols (ie @) in tag attributes

It also may be that an argument in an HTML tag uses characters that can't be used in python arguments. To handle these you can define those args using a dictionary.

```
Div(normal_arg='normal stuff',**{'notNormal:arg:with_varing@symbols!':'123'})
```

```
<div normal-arg="normal stuff" notnormal:arg:with_varing@symbols!="123"></div>
```

 Report an issue



 Reference > Handling handlers

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.

```
@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.)

```
@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 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_=eyJhdXRoIjogIjIwMjQ1MDk0MDc0MDY6NTY6MjAuNDQ1OTUwIiwgIm5hbWUiOiAiMiJ9.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



Live Reloading

When building your app it can be useful to view your changes in a web browser as you make them. FastHTML supports live reloading which means that it watches for any changes to your code and automatically refreshes the webpage in your browser.

To enable live reloading simply replace [FastHTML](#) in your app with [FastHTMLWithLiveReload](#).

```
from fasthtml.common import *  
app = FastHTMLWithLiveReload()
```

Then in your terminal run [uvicorn](#) with reloading enabled.

```
uvicorn main:app --reload
```

⚠ Gotchas - A reload is only triggered when you save your changes. - [FastHTMLWithLiveReload](#) should only be used during development. - If your app spans multiple directories you might need to use the [--reload-dir](#) flag to watch all files in each directory. See the [uvicorn docs](#) for more info. - The live reload script is only injected into the page when rendering [ft components](#).

Live reloading with [fast_app](#)

In development the [fast_app](#) function provides the same functionality. It instantiates the [FastHTMLWithLiveReload](#) class if you pass [live=True](#):


```
main.py  
  
from fasthtml.common import *  
  
app, rt = fast_app(live=True)  
  
serve()
```

①

②

- ① [fast_app\(\)](#) instantiates the [FastHTMLWithLiveReload](#) class.
- ② [serve\(\)](#) is a wrapper around a [uvicorn](#) call.

To run [main.py](#) in live reload mode, just do [python main.py](#). We recommend turning off live reload when deploying your app to production.

 [Report an issue](#)



 Source > fastapp

fastapp

The `fast_app` convenience wrapper

Usage can be summarized as:

```
from fasthtml.common import *

app, rt = fast_app()

@rt
def index(): return Titled("A demo of fast_app()@")

serve()
```

fast_app

[source](#)

```
fast_app (db_file:Optional[str]=None, render:Optional[<built-
infunctioncallable>]=None, hdrs:Optional[tuple]=None,
ftrs:Optional[tuple]=None, tbls:Optional[dict]=None,
before:Union[tuple,NoneType,fasthtml.core.Beforeware]=None,
middleware:Optional[tuple]=None, live:bool=False,
debug:bool=False, routes:Optional[tuple]=None,
exception_handlers:Optional[dict]=None,
on_startup:Optional[<built-infunctioncallable>]=None,
on_shutdown:Optional[<built-infunctioncallable>]=None,
lifespan:Optional[<built-infunctioncallable>]=None,
default_hdrs=True, pico:Optional[bool]=None,
surreal:Optional[bool]=True, htmx:Optional[bool]=True,
ws_hdr:bool=False, secret_key:Optional[str]=None,
key_fname:str='.sesskey', session_cookie:str='session_',
max_age:int=31536000, sess_path:str='/', same_site:str='lax',
sess_https_only:bool=False, sess_domain:Optional[str]=None,
htmlkw:Optional[dict]=None, bodykw:Optional[dict]=None,
reload_attempts:Optional[int]=1,
reload_interval:Optional[int]=1000, static_path:str='.',
**kwargs)
```

Create a *FastHTML* or *FastHTMLWithLiveReload* app.

	Type	Default	Details
db_file	Optional	None	Database file name, if needed

	Type	Default	Details
render	Optional	None	Function used to render default database class
hdrs	Optional	None	Additional FT elements to add to
fters	Optional	None	Additional FT elements to add to end of
tbls	Optional	None	Experimental mapping from DB table names to dict table definitions
before	Union	None	Functions to call prior to calling handler
middleware	Optional	None	Standard Starlette middleware
live	bool	False	Enable live reloading
debug	bool	False	Passed to Starlette, indicating if debug tracebacks should be returned on errors
routes	Optional	None	Passed to Starlette
exception_handlers	Optional	None	Passed to Starlette
on_startup	Optional	None	Passed to Starlette
on_shutdown	Optional	None	Passed to Starlette
lifespan	Optional	None	Passed to Starlette
default_hdrs	bool	True	Include default FastHTML headers such as HTMX script?
pico	Optional	None	Include PicoCSS header?
surreal	Optional	True	Include surreal.js/scope headers?
htmx	Optional	True	Include HTMX header?
ws_hdr	bool	False	Include HTMX websocket extension header?
secret_key	Optional	None	Signing key for sessions

	Type	Default	Details
key_fname	str	.sesskey	Session cookie signing key file name
session_cookie	str	session_	Session cookie name
max_age	int	31536000	Session cookie expiry time
sess_path	str	/	Session cookie path
same_site	str	lax	Session cookie same site policy
sess_https_only	bool	False	Session cookie HTTPS only?
sess_domain	Optional	None	Session cookie domain
htmlkw	Optional	None	Attrs to add to the HTML tag
bodykw	Optional	None	Attrs to add to the Body tag
reload_attempts	Optional	1	Number of reload attempts when live reloading
reload_interval	Optional	1000	Time between reload attempts in ms
static_path	str	.	Where the static file route points to, defaults to root dir
kwargs			
Returns	Any		

[!\[\]\(99f58673407353e96a019fbca558fd72_img.jpg\) Report an issue](#)



 Source > Pico.css components

Pico.css components

Basic components for generating Pico CSS tags

`picocondlink` is the class-conditional css `link` tag, and `picolink` is the regular tag.

```
show(picocondlink)
```

set_pico_cls

[source](#)

```
set_pico_cls ()
```

Run this to make jupyter outputs styled with pico:

```
set_pico_cls()
```

Card

[source](#)

```
Card (*c, header=None, footer=None, target_id=None, hx_vals=None,
      id=None, cls=None, title=None, style=None, accesskey=None,
      contenteditable=None, dir=None, draggable=None, enterkeyhint=None,
      hidden=None, inert=None, inputmode=None, lang=None, popover=None,
      spellcheck=None, tabindex=None, translate=None, hx_get=None,
      hx_post=None, hx_put=None, hx_delete=None, hx_patch=None,
      hx_trigger=None, hx_target=None, hx_swap=None, hx_swap_oob=None,
      hx_include=None, hx_select=None, hx_select_oob=None,
      hx_indicator=None, hx_push_url=None, hx_confirm=None,
      hx_disable=None, hx_replace_url=None, hx_disabled_elt=None,
      hx_ext=None, hx_headers=None, hx_history=None, hx_history_elt=None,
      hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None,
      hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

A PicoCSS Card, implemented as an Article with optional Header and Footer

```
show(Card('body', header=P('head'), footer=P('foot')))
```

head

body

foot

Group

[source](#)

```
Group (*c, target_id=None, hx_vals=None, id=None, cls=None, title=None,
      style=None, accesskey=None, contenteditable=None, dir=None,
      draggable=None, enterkeyhint=None, hidden=None, inert=None,
      inputmode=None, lang=None, popover=None, spellcheck=None,
      tabindex=None, translate=None, hx_get=None, hx_post=None,
      hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
      hx_target=None, hx_swap=None, hx_swap_oob=None, hx_include=None,
      hx_select=None, hx_select_oob=None, hx_indicator=None,
      hx_push_url=None, hx_confirm=None, hx_disable=None,
      hx_replace_url=None, hx_disabled_elt=None, hx_ext=None,
      hx_headers=None, hx_history=None, hx_history_elt=None,
      hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None,
      hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

A PicoCSS Group, implemented as a Fieldset with role 'group'

```
show(Group(Input(), Button("Save")))
```

Save

Search

[source](#)

```
Search (*c, target_id=None, hx_vals=None, id=None, cls=None, title=None,
       style=None, accesskey=None, contenteditable=None, dir=None,
       draggable=None, enterkeyhint=None, hidden=None, inert=None,
       inputmode=None, lang=None, popover=None, spellcheck=None,
       tabindex=None, translate=None, hx_get=None, hx_post=None,
       hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
       hx_target=None, hx_swap=None, hx_swap_oob=None, hx_include=None,
       hx_select=None, hx_select_oob=None, hx_indicator=None,
       hx_push_url=None, hx_confirm=None, hx_disable=None,
       hx_replace_url=None, hx_disabled_elt=None, hx_ext=None,
       hx_headers=None, hx_history=None, hx_history_elt=None,
       hx_inherit=None, hx_params=None, hx_preserve=None,
       hx_prompt=None, hx_request=None, hx_sync=None, hx_validate=None,
       **kwargs)
```

A PicoCSS Search, implemented as a Form with role 'search'

```
show(Search(Input(type="search"), Button("Search")))
```

Grid

[source](#)

```
Grid (*c, cls='grid', target_id=None, hx_vals=None, id=None, title=None,
     style=None, accesskey=None, contenteditable=None, dir=None,
     draggable=None, enterkeyhint=None, hidden=None, inert=None,
     inputmode=None, lang=None, popover=None, spellcheck=None,
     tabindex=None, translate=None, hx_get=None, hx_post=None,
     hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
     hx_target=None, hx_swap=None, hx_swap_oob=None, hx_include=None,
     hx_select=None, hx_select_oob=None, hx_indicator=None,
     hx_push_url=None, hx_confirm=None, hx_disable=None,
     hx_replace_url=None, hx_disabled_elt=None, hx_ext=None,
     hx_headers=None, hx_history=None, hx_history_elt=None,
     hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None,
     hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

A PicoCSS Grid, implemented as child Divs in a Div with class 'grid'

```
colors = [Input(type="color", value=o) for o in ('#e66465', '#53d2c5', '#f6b73c')]
show(Grid(*colors))
```

DialogX

[source](#)

```
DialogX (*c, open=None, header=None, footer=None, id=None,
         target_id=None, hx_vals=None, cls=None, title=None, style=None,
         accesskey=None, contenteditable=None, dir=None, draggable=None,
         enterkeyhint=None, hidden=None, inert=None, inputmode=None,
         lang=None, popover=None, spellcheck=None, tabindex=None,
         translate=None, hx_get=None, hx_post=None, hx_put=None,
         hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
         hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
         hx_select_oob=None, hx_indicator=None, hx_push_url=None,
         hx_confirm=None, hx_disable=None, hx_replace_url=None,
         hx_disabled_elt=None, hx_ext=None, hx_headers=None,
         hx_history=None, hx_history_elt=None, hx_inherit=None,
```

```
hx_params=None, hx_preserve=None, hx_prompt=None,
hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

A PicoCSS Dialog, with children inside a Card

```
hdr = Div(Button(aria_label="Close", rel="prev"), P('confirm'))
ftr = Div(Button('Cancel', cls="secondary"), Button('Confirm'))
d = DialogX('thank you!', header=hdr, footer=ftr, open=None, id='dlgtest')
# use js or htmx to display modal
```

Container

[source](#)

```
Container (*args, target_id=None, hx_vals=None, id=None, cls=None,
title=None, style=None, accesskey=None, contenteditable=None,
dir=None, draggable=None, enterkeyhint=None, hidden=None,
inert=None, inputmode=None, lang=None, popover=None,
spellcheck=None, tabindex=None, translate=None, hx_get=None,
hx_post=None, hx_put=None, hx_delete=None, hx_patch=None,
hx_trigger=None, hx_target=None, hx_swap=None,
hx_swap_oob=None, hx_include=None, hx_select=None,
hx_select_oob=None, hx_indicator=None, hx_push_url=None,
hx_confirm=None, hx_disable=None, hx_replace_url=None,
hx_disabled_elt=None, hx_ext=None, hx_headers=None,
hx_history=None, hx_history_elt=None, hx_inherit=None,
hx_params=None, hx_preserve=None, hx_prompt=None,
hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

A PicoCSS Container, implemented as a Main with class 'container'

PicoBusy

[source](#)

```
PicoBusy ()
```

 Report an issue

Core

The **FastHTML** subclass of **Starlette**, along with the **RouterX** and **RouteX** classes it automatically uses.

This is the source code to fasthtml. You won't need to read this unless you want to understand how things are built behind the scenes, or need full details of a particular API. The notebook is converted to the Python module [fasthtml/core.py](#) using [nbdev](#).

Imports and utils

```
import time

from IPython import display
from enum import Enum
from pprint import pprint

from fastcore.test import *
from starlette.testclient import TestClient
from starlette.requests import Headers
from starlette.datastructures import UploadFile
```

We write source code *first*, and then tests come *after*. The tests serve as both a means to confirm that the code works and also serves as working examples. The first declared function, **date**, is an example of this pattern.

parsed_date

[source](#)

```
parsed_date (s:str)
```

Convert **s** to a datetime

```
parsed_date('2pm')
datetime.datetime(2024, 10, 16, 14, 0)
```

```
isinstance(date.fromtimestamp(0), date)
True
```

snake2hyphens

[source](#)

```
snake2hyphens (s:str)
```

Convert **s** from snake case to hyphenated and capitalised

```
snake2hyphens("snake_case")
```

```
'Snake-Case'
```

HtmxHeaders

[source](#)

```
HtmxHeaders (boosted:str|None=None, current_url:str|None=None,
             history_restore_request:str|None=None, prompt:str|None=None,
             request:str|None=None, target:str|None=None,
             trigger_name:str|None=None, trigger:str|None=None)
```

```
def test_request(url: str='/', headers: dict={}, method: str='get') -> Request:
    scope = {
        'type': 'http',
        'method': method,
        'path': url,
        'headers': Headers(headers).raw,
        'query_string': b'',
        'scheme': 'http',
        'client': ('127.0.0.1', 8000),
        'server': ('127.0.0.1', 8000),
    }
    receive = lambda: {"body": b"", "more_body": False}
    return Request(scope, receive)
```

```
h = test_request(headers=Headers({'HX-Request': '1'}))
_get_htmx(h.headers)
```

```
HtmxHeaders(boosted=None, current_url=None, history_restore_request=None,
prompt=None, request='1', target=None, trigger_name=None, trigger=None)
```

Request and response

```
test_eq(_fix_anno(Union[str,None])('a'), 'a')
test_eq(_fix_anno(float)(0.9), 0.9)
test_eq(_fix_anno(int>('1'), 1)
test_eq(_fix_anno(int)(['1','2']), 2)
test_eq(_fix_anno(list[int])(['1','2']), [1,2])
test_eq(_fix_anno(list[int])('1'), [1])
```

```
d = dict(k=int, l=List[int])
test_eq(_form_arg('k', "1", d), 1)
test_eq(_form_arg('l', "1", d), [1])
test_eq(_form_arg('l', ["1","2"], d), [1,2])
```


HTTPHeader

[source](#)

```
HTTPHeader (k:str, v:str)
```

```
_to_htmx_header('trigger_after_settle')
```

```
'HX-Trigger-After-Settle'
```

HtmxResponseHeaders

[source](#)

```
HtmxResponseHeaders (location=None, push_url=None, redirect=None,
                      refresh=None, replace_url=None, reswap=None,
                      retarget=None, reselect=None, trigger=None,
                      trigger_after_settle=None, trigger_after_swap=None)
```

HTMX response headers

```
HtmxResponseHeaders(trigger_after_settle='hi')
```

```
HTTPHeader(k='HX-Trigger-After-Settle', v='hi')
```

form2dict

[source](#)

```
form2dict (form:starlette.datastructures.FormData)
```

Convert starlette form data to a dict

```
d = [('a',1),('a',2),('b',0)]
fd = FormData(d)
res = form2dict(fd)
test_eq(res['a'], [1,2])
test_eq(res['b'], 0)
```

parse_form

[source](#)

```
parse_form (req:starlette.requests.Request)
```

Starlette errors on empty multipart forms, so this checks for that situation

```
async def f(req):
    def _f(p:HTTPHeader): ...
    p = first(_sig(_f).parameters.values())
    result = await _from_body(req, p)
    return JSONResponse(result.__dict__)

app = Starlette(routes=[Route('/', f, methods=['POST'])])
client = TestClient(app)
```

```
d = dict(k='value1',v=['value2','value3'])
response = client.post('/', data=d)
print(response.json())
```

```
{'k': 'value1', 'v': 'value3'}
```

```
async def f(req): return Response(str(req.query_params.getlist('x')))
app = Starlette(routes=[Route('/', f, methods=['GET'])])
client = TestClient(app)
client.get('/?x=1&x=2').text
```

```
"['1', '2']"
```

```
def g(req, this:Starlette, a:str, b:HttpHeader): ...
```

```
async def f(req):
    a = await _wrap_req(req, _sig(g).parameters)
    return Response(str(a))
```

```
app = Starlette(routes=[Route('/', f, methods=['POST'])])
client = TestClient(app)
```

```
response = client.post('/?a=1', data=d)
print(response.text)
```

```
[<starlette.requests.Request object>, <starlette.applications.Starlette object>,
'1', HttpHeader(k='value1', v='value3')]
```

```
def g(req, this:Starlette, a:str, b:HttpHeader): ...
```

```
async def f(req):
    a = await _wrap_req(req, _sig(g).parameters)
    return Response(str(a))
```

```
app = Starlette(routes=[Route('/', f, methods=['POST'])])
client = TestClient(app)
```

```
response = client.post('/?a=1', data=d)
print(response.text)
```

```
[<starlette.requests.Request object>, <starlette.applications.Starlette object>,
'1', HttpHeader(k='value1', v='value3')]
```

flat_xt

[source](#)

```
flat_xt (lst)
```

Flatten lists

```
x = ft('a',1)
test_eq(flat_xt([x, x, [x,x]]), (x,)*4)
test_eq(flat_xt(x), (x,))
```

Beforeware

Beforeware (f, skip=None)

Initialize self. See help(type(self)) for accurate signature.

Websockets / SSE

```
def on_receive(self, msg:str): return f"Message text was: {msg}"
c = _ws_endp(on_receive)
app = Starlette(routes=[WebSocketRoute('/', _ws_endp(on_receive))])

cli = TestClient(app)
with cli.websocket_connect('/') as ws:
    ws.send_text('{"msg":"Hi!"}')
    data = ws.receive_text()
    assert data == 'Message text was: Hi!'
```

EventStream

EventStream (s)

Create a text/event-stream response from [s](#)

signal_shutdown

signal_shutdown ()

Routing and application

WS_RouteX

WS_RouteX (app, path:str, recv, conn:<built-infunctioncallable>=None,
disconn:<built-infunctioncallable>=None, name=None,
middleware=None)

Initialize self. See help(type(self)) for accurate signature.

uri

uri (_arg, **kwargs)

decode_uri

[source](#)

```
decode_uri (s)
```

StringConvertor.to_string

[source](#)

```
StringConvertor.to_string (value:str)
```

HTTPConnection.url_path_for

[source](#)

```
HTTPConnection.url_path_for (name:str, **path_params)
```

flat_tuple

[source](#)

```
flat_tuple (o)
```

Flatten lists

Redirect

[source](#)

```
Redirect (loc)
```

Use HTMX or Starlette RedirectResponse as required to redirect to [loc](#)

RouteX

[source](#)

```
RouteX (app, path:str, endpoint, methods=None, name=None,  
        include_in_schema=True, middleware=None)
```

Initialize self. See help(type(self)) for accurate signature.

RouterX

[source](#)

```
RouterX (app, routes=None, redirect_slashes=True, default=None,  
        middleware=None)
```

Initialize self. See help(type(self)) for accurate signature.

get_key

[source](#)

```
get_key (key=None, fname='.sesskey')
```

```
get_key()
```

```
'a604e4a2-08e8-462d-aff9-15468891fe09'
```

def_hdrs

[source](#)

```
def_hdrs (htmx=True, ct_hdr=False, ws_hdr=False, surreal=True)
```

Default headers for a FastHTML app

FastHTML

[source](#)

```
FastHTML (debug=False, routes=None, middleware=None,
          exception_handlers=None, on_startup=None, on_shutdown=None,
          lifespan=None, hdrs=None, ftrs=None, before=None, after=None,
          ws_hdr=False, ct_hdr=False, surreal=True, htmx=True,
          default_hdrs=True, sess_cls=<class
            'starlette.middleware.sessions.SessionMiddleware'>,
          secret_key=None, session_cookie='session_', max_age=31536000,
          sess_path='/', same_site='lax', sess_https_only=False,
          sess_domain=None, key_fname='.sesskey', htmlkw=None, **bodykw)
```

*Creates an application instance.

Parameters:

- **debug** - Boolean indicating if debug tracebacks should be returned on errors.
- **routes** - A list of routes to serve incoming HTTP and WebSocket requests.
- **middleware** - A list of middleware to run for every request. A starlette application will always automatically include two middleware classes. `ServerErrorMiddleware` is added as the very outermost middleware, to handle any uncaught errors occurring anywhere in the entire stack. `ExceptionMiddleware` is added as the very innermost middleware, to deal with handled exception cases occurring in the routing or endpoints.
- **exception_handlers** - A mapping of either integer status codes, or exception class types onto callables which handle the exceptions. Exception handler callables should be of the form `handler(request, exc) -> response` and may be either standard functions, or async functions.
- **on_startup** - A list of callables to run on application startup. Startup handler callables do not take any arguments, and may be either standard functions, or async functions.
- **on_shutdown** - A list of callables to run on application shutdown. Shutdown handler callables do not take any arguments, and may be either standard functions, or async functions.
- **lifespan** - A lifespan context function, which can be used to perform startup and shutdown tasks. This is a newer style that replaces the `on_startup` and `on_shutdown` handlers. Use one or the other, not both.*

FastHTML.route

[source](#)

```
FastHTML.route (path:str=None, methods=None, name=None,
                include_in_schema=True)
```

Add a route at [path](#)

serve

[source](#)

```
serve (appname=None, app='app', host='0.0.0.0', port=None, reload=True,
      reload_includes:list[str] | str | None=None,
      reload_excludes:list[str] | str | None=None)
```

Run the app in an async server, with live reload set as the default.

	Type	Default	Details
appname	NoneType	None	Name of the module
app	str	app	App instance to be served
host	str	0.0.0.0	If host is 0.0.0.0 will convert to localhost
port	NoneType	None	If port is None it will default to 5001 or the PORT environment variable
reload	bool	True	Default is to reload the app upon code changes
reload_includes	list[str] str None	None	Additional files to watch for changes
reload_excludes	list[str] str None	None	Files to ignore for changes

Client

[source](#)

```
Client (app, url='http://testserver')
```

A simple *httpx* ASGI client that doesn't require [async](#)

```
app = FastHTML(routes=[Route('/', lambda _: Response('test'))])
cli = Client(app)

cli.get('/').text

'test'
```

Note that you can also use Starlette's [TestClient](#) instead of FastHTML's [Client](#). They should be largely interchangeable.

FastHTML Tests

```
def get_cli(app): return app, TestClient(app), app.route
```

```
app, cli, rt = get_cli(FastHTML(secret_key='soopersecret'))
```

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

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

```
'Hi there'
```

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

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

```
'Postal'
```

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

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

```
'testserver'
```

```
@rt
def yoyo(): return 'a yoyo'

cli.post('/yoyo').text
```

```
'a yoyo'
```

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

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

```
@app.get
def autopost2(): return Html(Body(Div('Text.', cls='px-2', hx_post=show_host.rt(a='b'))
print(cli.get('/autopost2').text)
```

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

```
@app.get
def autoget2(): return Html(Div('Text.', hx_get=show_host))
print(cli.get('/autoget2').text)

<!doctype html>
<html>
  <div hx-get="/hostie">Text.</div>
</html>
```

```
@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!'
```

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

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

```
@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'
```

```
@app.get("/background")
async def background_task(request):
    async def long_running_task():
        await asyncio.sleep(0.1)
        print("Background task completed!")
    return P("Task started"), BackgroundTask(long_running_task)

response = cli.get("/background")

Background task completed!
```

```
test_eq(app.router.url_path_for('gday', nm='Jeremy'), '/user/Jeremy')
```

```
hxhdr = {'headers':{'hx-request':"1"}}

@rt('/ft')
def get(): return Title('Foo'),H1('bar')

txt = cli.get('/ft').text
assert '<title>Foo</title>' in txt and '<h1>bar</h1>' in txt and '<html>' in txt

@rt('/xt2')
def get(): return H1('bar')
```



```

txt = cli.get('/xt2').text
assert '<title>FastHTML page</title>' in txt and '<h1>bar</h1>' in txt and '<html>' in

assert cli.get('/xt2', **hxhdr).text.strip() == '<h1>bar</h1>'

@rt('/xt3')
def get(): return Html(Head(Title('hi')), Body(P('there')))

txt = cli.get('/xt3').text
assert '<title>FastHTML page</title>' not in txt and '<title>hi</title>' in txt and '<

```

```

@rt('/oops')
def get(nope): return nope
test_warns(lambda: cli.get('/oops?nope=1'))

```

```

def test_r(cli, path, exp, meth='get', hx=False, **kwargs):
    if hx: kwargs['headers'] = {'hx-request': "1"}
    test_eq(getattr(cli, meth)(path, **kwargs).text, exp)

ModelName = str_enum('ModelName', "alexnet", "resnet", "lenet")
fake_db = [{"name": "Foo"}, {"name": "Bar"}]

```

```

@rt('/html/{idx}')
async def get(idx: int): return Body(H4(f'Next is {idx+1}.'))

```

```

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

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

@rt("/items/")
def get(idx: int | None = 0): return fake_db[idx]

@rt("/idxl/")
def get(idx: list[int]): return str(idx)

```

```

r = cli.get('/html/1', headers={'hx-request': "1"})
assert '<h4>Next is 2.</h4>' in r.text
test_r(cli, '/models/alexnet', 'alexnet')
test_r(cli, '/files/foo', 'foo.txt')
test_r(cli, '/items/?idx=1', '{"name": "Bar"}')
test_r(cli, '/items/', '{"name": "Foo"}')
assert cli.get('/items/?idx=g').text == '404 Not Found'
assert cli.get('/items/?idx=g').status_code == 404
test_r(cli, '/idxl/?idx=1&idx=2', '[1, 2]')
assert cli.get('/idxl/?idx=1&idx=g').status_code == 404

```

```

app = FastHTML()
rt = app.route

```

```
cli = TestClient(app)
@app.route(r'/static/{path:path}.jpg')
def index(path:str): return f'got {path}'
cli.get('/static/sub/a.b.jpg').text

'got sub/a.b'
```

```
app.chk = 'foo'
```

```
@app.get("/booly/")
def _(coming:bool=True): return 'Coming' if coming else 'Not coming'

@app.get("/datie/")
def _(d:parsed_date): return d

@app.get("/ua")
async def _(user_agent:str): return user_agent

@app.get("/hxtest")
def _(htmx): return htmx.request

@app.get("/hxtest2")
def _(foo:HtmxHeaders, req): return foo.request

@app.get("/app")
def _(app): return app.chk

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

@app.get("/app3")
def _(foo:FastHTML): return HtmxResponseHeaders(location="http://example.org")

@app.get("/app4")
def _(foo:FastHTML): return Redirect("http://example.org")
```

```
test_r(cli, '/booly/?coming=true', 'Coming')
test_r(cli, '/booly/?coming=no', 'Not coming')
date_str = "17th of May, 2024, 2p"
test_r(cli, f'/datie/?d={date_str}', '2024-05-17 14:00:00')
test_r(cli, '/ua', 'FastHTML', headers={'User-Agent': 'FastHTML'})
test_r(cli, '/hxtest' , '1', headers={'HX-Request': '1'})
test_r(cli, '/hxtest2', '1', headers={'HX-Request': '1'})
test_r(cli, '/app' , 'foo')
```

```
r = cli.get('/app2', **hxhdr)
test_eq(r.text, 'foo')
test_eq(r.headers['mykey'], 'myval')
```

```
r = cli.get('/app3')
test_eq(r.headers['HX-Location'], 'http://example.org')
```

```
r = cli.get('/app4', follow_redirects=False)
test_eq(r.status_code, 303)
```

```
r = cli.get('/app4', headers={'HX-Request': '1'})
test_eq(r.headers['HX-Redirect'], 'http://example.org')
```

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

t = cli.post('/meta').text
assert re.search(r'<body>\s*<h1>hi</h1>\s*</body>', t)
assert '<meta' in t
```

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

test_r(cli, '/profile/me', 'Alexis', 'post', data={'username' : 'Alexis'})
test_r(cli, '/profile/me', 'Missing required field: username', 'post', data={})
```

```
# Example post request with parameter that has a default value
@app.post('/pet/dog')
def pet_dog(dogname: str = None): return dogname

# Working post request with optional parameter
test_r(cli, '/pet/dog', '', 'post', data={})
```

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

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

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

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

@app.post("/bodient/")
def bodient(data:nt): return asdict(data)

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

@app.post("/bodietd/")
def bodient(data:BodieTD): return data
```

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

@rt("/bodie2/", methods=['get','post'])
def bodie(d:Bodie2): return f"a: {d.a}; b: {d.b}"
```

```
from fasthtml.xtend import Titled
```

```
d = dict(a=1, b='foo')

test_r(cli, '/bodie/me', '{"a":1,"b":"foo","nm":"me"}', 'post', data=dict(a=1, b='foo')
test_r(cli, '/bodied/', '{"a":"1","b":"foo"}', 'post', data=d)
test_r(cli, '/bodie2/', 'a: 1; b: foo', 'post', data={'a':1})
test_r(cli, '/bodie2/?a=1&b=foo&nm=me', 'a: 1; b: foo')
test_r(cli, '/bodient/', '{"a":"1","b":"foo"}', 'post', data=d)
test_r(cli, '/bodietd/', '{"a":1,"b":"foo"}', 'post', data=d)
```

```
# Testing POST with Content-Type: application/json
@app.post("/")
def index(it: Bodie): return Titled("It worked!", P(f"{it.a}, {it.b}"))

s = json.dumps({"b": "Lorem", "a": 15})
response = cli.post('/', headers={"Content-Type": "application/json"}, data=s).text
assert "<title>It worked!</title>" in response and "<p>15, Lorem</p>" in response
```

```
# Testing POST with Content-Type: application/json
@app.post("/bodytext")
def index(body): return body

response = cli.post('/bodytext', headers={"Content-Type": "application/json"}, data=s)
test_eq(response, '{"b": "Lorem", "a": 15}')
```

```
files = [ ('files', ('file1.txt', b'content1')),
          ('files', ('file2.txt', b'content2')) ]
```

```
@rt("/uploads")
async def post(files:list[UploadFile]):
    return ','.join([await file.read().decode() for file in files])

res = cli.post('/uploads', files=files)
print(res.status_code)
print(res.text)
```

```
200
content1,content2
```

```
@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 2024-10-16 15:38:25.588198
'Session time: 2024-10-16 15:38:25.588198'

```

```

@rt("/sess-first")
def post(sess, name: str):
    sess["name"] = name
    return str(sess)

cli.post('/sess-first', data={'name': 2})

@rt("/getsess-all")
def get(sess): return sess['name']

test_eq(cli.get('/getsess-all').text, '2')

```

```

@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

```

```

@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)

```

```

h = cli.options('/form-submit/2').headers
test_eq(h['Access-Control-Allow-Methods'], 'POST')

```

```

from fasthtml.authmw import user_pwd_auth

```

```

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

app,cli,rt = get_cli(FastHTML(exception_handlers={404:_not_found}))

txt = cli.get('/').text

```

```
assert '<div>nope</div>' in txt
assert '<!doctype html>' in txt
```

```
app,cli,rt = get_cli(FastHTML())

@rt("/{name}/{age}")
def get(name: str, age: int):
    return Titled(f"Hello {name.title()}, age {age}")

assert '<title>Hello Uma, age 5</title>' in cli.get('/uma/5').text
assert '404 Not Found' in cli.get('/uma/five').text
```

```
auth = user_pwd_auth(testuser='spycraft')
app,cli,rt = get_cli(FastHTML(middleware=[auth]))

@rt("/locked")
def get(auth): return 'Hello, ' + auth

test_eq(cli.get('/locked').text, 'not authenticated')
test_eq(cli.get('/locked', auth=("testuser","spycraft")).text, 'Hello, testuser')
```

```
auth = user_pwd_auth(testuser='spycraft')
app,cli,rt = get_cli(FastHTML(middleware=[auth]))

@rt("/locked")
def get(auth): return 'Hello, ' + auth

test_eq(cli.get('/locked').text, 'not authenticated')
test_eq(cli.get('/locked', auth=("testuser","spycraft")).text, 'Hello, testuser')
```

Extras

```
app,cli,rt = get_cli(FastHTML(secret_key='soopersecret'))
```

[source](#)

cookie

```
cookie (key:str, value='', max_age=None, expires=None, path='/',
        domain=None, secure=False, httponly=False, samesite='lax')
```

Create a 'set-cookie' [HttpHeader](#)

```
@rt("/setcookie")
def get(req): 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 15:38:25.962118'
```

reg_re_param

[source](#)

```
reg_re_param (m, s)
```

FastHTML.static_route_exts

[source](#)

```
FastHTML.static_route_exts (prefix='/', static_path='.', exts='static')
```

Add a static route at URL path *prefix* with files from *static_path* and *exts* defined by *reg_re_param()*

```
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}"

test_r(cli, '/static/foo/jph.me.ico', 'Getting jph.me.ico from /foo/')
```

```
app.static_route_exts()
assert 'These are the source notebooks for FastHTML' in cli.get('/README.txt').text
```

FastHTML.static_route

[source](#)

```
FastHTML.static_route (ext='', prefix='/', static_path='.')
```

Add a static route at URL path *prefix* with files from *static_path* and single *ext* (including the '?')

```
app.static_route('.md', static_path='../..')
assert 'THIS FILE WAS AUTOGENERATED' in cli.get('/README.md').text
```

MiddlewareBase

[source](#)

```
MiddlewareBase ()
```

Initialize self. See *help(type(self))* for accurate signature.

FtResponse

[source](#)

```
FtResponse (content, status_code:int=200, headers=None, cls=<class
    'starlette.responses.HTMLResponse'>,
```

```
media_type:str|None=None)
```

Wrap an *FT* response with any Starlette *Response*

```
@rt('/ftr')
def get():
    cts = Title('Foo'),H1('bar')
    return FtResponse(cts, status_code=201, headers={'Location':'/foo/1'})

r = cli.get('/ftr')

test_eq(r.status_code, 201)
test_eq(r.headers['location'], '/foo/1')
txt = r.text
assert '<title>Foo</title>' in txt and '<h1>bar</h1>' in txt and '<html>' in txt
```

unqid


[source](#)

```
unqid ()
```

setup_ws

[source](#)

```
setup_ws (app)
```

 Report an issue

[Source](#) > Components

Components

`ft_html` and `ft_hx` functions to add some conveniences to `ft`, along with a full set of basic HTML components, and functions to work with forms and `FT` conversion

```
from lxml import html as lx
from pprint import pprint
```

show

[source](#)

```
show (ft, *rest)
```

Renders FT Components into HTML within a Jupyter notebook.

```
sentence = P(Strong("FastHTML is ", I("Fast")))

# When placed within the `show()` function, this will render
# the HTML in Jupyter notebooks.
show(sentence)
```

FastHTML is *Fast*

```
# Called without the `show()` function, the raw HTML is displayed
sentence
```

```
<p>
<strong>FastHTML is <i>Fast</i></strong></p>
```

attrmap_x

[source](#)

```
attrmap_x (o)
```

ft_html

[source](#)

```
ft_html (tag:str, *c, id=None, cls=None, title=None, style=None,
         attrmap=None, valmap=None, ft_cls=None, auto_id=None, **kwargs)
```

ft_hx

```
ft_hx (tag:str, *c, target_id=None, hx_vals=None, id=None, cls=None,
      title=None, style=None, accesskey=None, contenteditable=None,
      dir=None, draggable=None, enterkeyhint=None, hidden=None,
      inert=None, inputmode=None, lang=None, popover=None,
      spellcheck=None, tabindex=None, translate=None, hx_get=None,
      hx_post=None, hx_put=None, hx_delete=None, hx_patch=None,
      hx_trigger=None, hx_target=None, hx_swap=None, hx_swap_oob=None,
      hx_include=None, hx_select=None, hx_select_oob=None,
      hx_indicator=None, hx_push_url=None, hx_confirm=None,
      hx_disable=None, hx_replace_url=None, hx_disabled_elt=None,
      hx_ext=None, hx_headers=None, hx_history=None,
      hx_history_elt=None, hx_inherit=None, hx_params=None,
      hx_preserve=None, hx_prompt=None, hx_request=None, hx_sync=None,
      hx_validate=None, **kwargs)
```

```
ft_html('a', _at_click_dot_away=1)
```

```
<a @click_dot_away="1"></a>
```

```
ft_html('a', **{'@click.away':1})
```

```
<a @click.away="1"></a>
```

```
ft_html('a', {'@click.away':1})
```

```
<a @click.away="1"></a>
```

```
ft_hx('a', hx_vals={'a':1})
```

```
<a hx-vals='{"a": 1}'></a>
```

File

File (fname)

Use the unescaped text in file *fname* directly

For tags that have a *name* attribute, it will be set to the value of *id* if not provided explicitly:

```
Form(Button(target_id='foo', id='btn'),
      hx_post='/', target_id='tgt', id='frm')
```

```
<form hx-post="/" hx-target="#tgt" id="frm" name="frm"><button hx-target="#foo" id="t
```

fill_form

[source](#)

```
fill_form (form:fastcore.xml.FT, obj)
```

Fills named items in *form* using attributes in *obj*

```
@dataclass
class TodoItem:
    title:str; id:int; done:bool; details:str; opt:str='a'

todo = TodoItem(id=2, title="Profit", done=True, details="Details", opt='b')
check = Label(Input(type="checkbox", cls="checkboxer", name="done", data_foo="bar"), "
form = Form(Fieldset(Input(cls="char", id="title", value="a"), check, Input(type="hidden",
                        Select(Option(value='a'), Option(value='b'), name='opt'),
                        Textarea(id='details'), Button("Save"),
                        name="stuff"))
form = fill_form(form, todo)
assert '<textarea id="details" name="details">Details</textarea>' in to_xml(form)
form
```

```
<form><fieldset name="stuff">    <input value="Profit" id="title" class="char" name='
<label class="px-2">        <input type="checkbox" name="done" data-foo="bar" class="ch
Done</label>    <input type="hidden" id="id" name="id" value="2">
<select name="opt"><option value="a"></option><option value="b" selected="1"></option>
```

fill_dataclass

[source](#)

```
fill_dataclass (src, dest)
```

Modifies dataclass in-place and returns it

```
nt = TodoItem('', 0, False, '')
fill_dataclass(todo, nt)
nt
```

```
TodoItem(title='Profit', id=2, done=True, details='Details', opt='b')
```

find_inputs

[source](#)

```
find_inputs (e, tags='input', **kw)
```

Recursively find all elements in *e* with *tags* and attrs matching *kw*

```
inps = find_inputs(form, id='title')
test_eq(len(inps), 1)
inps
```

```
[input(),{'value': 'Profit', 'id': 'title', 'class': 'char', 'name': 'title'}]]
```

You can also use `lxml` for more sophisticated searching:

```
elem = lx.fromstring(to_xml(form))
test_eq(elem.xpath("//input[@id='title']/@value"), ['Profit'])
```

[source](#)

getattr

```
__getattr__ (tag)
```

[source](#)

html2ft

```
html2ft (html, attr1st=False)
```

Convert HTML to an *ft* expression

```
h = to_xml(form)
hl_md(html2ft(h), 'python')
```

```
Form(
  Fieldset(
    Input(value='Profit', id='title', name='title', cls='char'),
    Label(
      Input(type='checkbox', name='done', data_foo='bar', checked='1', cls='checkbox'),
      'Done',
      cls='px-2'
    ),
    Input(type='hidden', id='id', name='id', value='2'),
    Select(
      Option(value='a'),
      Option(value='b', selected='1'),
      name='opt'
    ),
    Textarea('Details', id='details', name='details'),
    Button('Save'),
    name='stuff'
  )
)
```

```
hl_md(html2ft(h, attr1st=True), 'python')
```

```
Form(
  Fieldset(name='stuff')(
    Input(value='Profit', id='title', name='title', cls='char'),
    Label(cls='px-2')(
      Input(type='checkbox', name='done', data_foo='bar', checked='1', cls='checkbox'),
      'Done'
    ),
    Input(type='hidden', id='id', name='id', value='2'),
    Select(name='opt')(
```

```
        Option(value='a'),
        Option(value='b', selected='1')
    ),
    Textarea('Details', id='details', name='details'),
    Button('Save')
)
)
```

sse_message


[source](#)

```
sse_message (elm, event='message')
```

Convert element *elm* into a format suitable for SSE streaming

```
print(sse_message(Div(P('hi'), P('there'))))
```

```
event: message
data: <div>
data:   <p>hi</p>
data:   <p>there</p>
data: </div>
```

 [Report an issue](#)



 Source > Component extensions

Component extensions

Simple extensions to standard HTML components, such as adding sensible defaults

```
from pprint import pprint
```

A

[source](#)

```
A (*c, hx_get=None, target_id=None, hx_swap=None, href='#', hx_vals=None,
   id=None, cls=None, title=None, style=None, accesskey=None,
   contenteditable=None, dir=None, draggable=None, enterkeyhint=None,
   hidden=None, inert=None, inputmode=None, lang=None, popover=None,
   spellcheck=None, tabindex=None, translate=None, hx_post=None,
   hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
   hx_target=None, hx_swap_oob=None, hx_include=None, hx_select=None,
   hx_select_oob=None, hx_indicator=None, hx_push_url=None,
   hx_confirm=None, hx_disable=None, hx_replace_url=None,
   hx_disabled_elt=None, hx_ext=None, hx_headers=None, hx_history=None,
   hx_history_elt=None, hx_inherit=None, hx_params=None,
   hx_preserve=None, hx_prompt=None, hx_request=None, hx_sync=None,
   hx_validate=None, **kwargs)
```

An A tag: *href* defaults to '#' for more concise use with HTMX

```
A('text', ht_get='/get', target_id='id')
```

```
<a href="#" ht-get="/get" hx-target="#id">text</a>
```

AX

[source](#)

```
AX (txt, hx_get=None, target_id=None, hx_swap=None, href='#',
    hx_vals=None, id=None, cls=None, title=None, style=None,
    accesskey=None, contenteditable=None, dir=None, draggable=None,
    enterkeyhint=None, hidden=None, inert=None, inputmode=None,
    lang=None, popover=None, spellcheck=None, tabindex=None,
    translate=None, hx_post=None, hx_put=None, hx_delete=None,
    hx_patch=None, hx_trigger=None, hx_target=None, hx_swap_oob=None,
    hx_include=None, hx_select=None, hx_select_oob=None,
    hx_indicator=None, hx_push_url=None, hx_confirm=None,
    hx_disable=None, hx_replace_url=None, hx_disabled_elt=None,
    hx_ext=None, hx_headers=None, hx_history=None, hx_history_elt=None,
```

```
hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None,
hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

An `A` tag with just one text child, allowing `hx_get`, `target_id`, and `hx_swap` to be positional params

```
AX('text', '/get', 'id')
```

```
<a href="#" hx-get="/get" hx-target="#id">text</a>
```

Forms

Form

[source](#)

```
Form (*c, enctype='multipart/form-data', target_id=None, hx_vals=None,
      id=None, cls=None, title=None, style=None, accesskey=None,
      contenteditable=None, dir=None, draggable=None, enterkeyhint=None,
      hidden=None, inert=None, inputmode=None, lang=None, popover=None,
      spellcheck=None, tabindex=None, translate=None, hx_get=None,
      hx_post=None, hx_put=None, hx_delete=None, hx_patch=None,
      hx_trigger=None, hx_target=None, hx_swap=None, hx_swap_oob=None,
      hx_include=None, hx_select=None, hx_select_oob=None,
      hx_indicator=None, hx_push_url=None, hx_confirm=None,
      hx_disable=None, hx_replace_url=None, hx_disabled_elt=None,
      hx_ext=None, hx_headers=None, hx_history=None, hx_history_elt=None,
      hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None,
      hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

A `Form` tag; identical to plain [ft_hx](#) version except default `enctype='multipart/form-data'`

Hidden

[source](#)

```
Hidden (value:Any='', id:Any=None, target_id=None, hx_vals=None,
        cls=None, title=None, style=None, accesskey=None,
        contenteditable=None, dir=None, draggable=None,
        enterkeyhint=None, hidden=None, inert=None, inputmode=None,
        lang=None, popover=None, spellcheck=None, tabindex=None,
        translate=None, hx_get=None, hx_post=None, hx_put=None,
        hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
        hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
        hx_select_oob=None, hx_indicator=None, hx_push_url=None,
        hx_confirm=None, hx_disable=None, hx_replace_url=None,
        hx_disabled_elt=None, hx_ext=None, hx_headers=None,
        hx_history=None, hx_history_elt=None, hx_inherit=None,
        hx_params=None, hx_preserve=None, hx_prompt=None,
        hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

An `Input` of type `'hidden'`

CheckboxX

```
CheckboxX (checked:bool=False, label=None, value='1', id=None, name=None,
          target_id=None, hx_vals=None, cls=None, title=None,
          style=None, accesskey=None, contenteditable=None, dir=None,
          draggable=None, enterkeyhint=None, hidden=None, inert=None,
          inputmode=None, lang=None, popover=None, spellcheck=None,
          tabindex=None, translate=None, hx_get=None, hx_post=None,
          hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
          hx_target=None, hx_swap=None, hx_swap_oob=None,
          hx_include=None, hx_select=None, hx_select_oob=None,
          hx_indicator=None, hx_push_url=None, hx_confirm=None,
          hx_disable=None, hx_replace_url=None, hx_disabled_elt=None,
          hx_ext=None, hx_headers=None, hx_history=None,
          hx_history_elt=None, hx_inherit=None, hx_params=None,
          hx_preserve=None, hx_prompt=None, hx_request=None,
          hx_sync=None, hx_validate=None, **kwargs)
```

A Checkbox optionally inside a Label, preceded by a [Hidden](#) with matching name

```
show(CheckboxX(True, 'Check me out!'))
```

☒ Check me out!

Script

[source](#)

```
Script (code:str='', id=None, cls=None, title=None, style=None,
        attrmap=None, valmap=None, ft_cls=None, auto_id=None, **kwargs)
```

A Script tag that doesn't escape its code

Style

[source](#)

```
Style (*c, id=None, cls=None, title=None, style=None, attrmap=None,
       valmap=None, ft_cls=None, auto_id=None, **kwargs)
```

A Style tag that doesn't escape its code

Style and script templates

double_braces

[source](#)

```
double_braces (s)
```

Convert single braces to double braces if next to special chars or newline

undouble_braces

[source](#)

```
undouble_braces (s)
```

Convert double braces to single braces if next to special chars or newline

loose_format

[source](#)

```
loose_format (s, **kw)
```

String format *s* using *kw*, without being strict about braces outside of template params

ScriptX

[source](#)

```
ScriptX (fname, src=None, nomodule=None, type=None, _async=None,
         defer=None, charset=None, crossorigin=None, integrity=None,
         **kw)
```

A *script* element with contents read from *fname*

replace_css_vars

[source](#)

```
replace_css_vars (css, pre='tpl', **kwargs)
```

Replace *var(--)* CSS variables with *kwargs* if name prefix matches *pre*

StyleX

[source](#)

```
StyleX (fname, **kw)
```

A *style* element with contents read from *fname* and variables replaced from *kw*

Nbsp

[source](#)

```
Nbsp ()
```

A non-breaking space

Surreal and JS

Surreal

[source](#)

```
Surreal (code:str)
```

Wrap *code* in *domReadyExecute* and set *m=me()* and *p=me('-')*

On

[source](#)

```
On (code:str, event:str='click', sel:str='', me=True)
```

An async *surreal.js* script block event handler for *event* on selector *sel,p*, making available parent *p*, event *ev*, and target *e*

Prev

[source](#)

```
Prev (code:str, event:str='click')
```

An async *surreal.js* script block event handler for *event* on previous sibling, with same vars as [On](#)

Now

[source](#)

```
Now (code:str, sel:str='')
```

An async *surreal.js* script block on selector *me(sel)*

AnyNow

[source](#)

```
AnyNow (sel:str, code:str)
```

An async *surreal.js* script block on selector *any(sel)*

run_js

[source](#)

```
run_js (js, id=None, **kw)
```

Run *js* script, auto-generating *id* based on name of caller if needed, and js-escaping any *kw* params

HtmxOn

[source](#)

```
HtmxOn (eventname:str, code:str)
```

jsd

[source](#)

```
jsd (org, repo, root, path, prov='gh', typ='script', ver=None, esm=False,
    **kwargs)
```

jsdelivr [Script](#) or CSS [Link](#) tag, or URL

Other helpers

Titled

[source](#)

```
Titled (title:str='FastHTML app', *args, cls='container', target_id=None,
    hx_vals=None, id=None, style=None, accesskey=None,
    contenteditable=None, dir=None, draggable=None,
    enterkeyhint=None, hidden=None, inert=None, inputmode=None,
    lang=None, popover=None, spellcheck=None, tabindex=None,
    translate=None, hx_get=None, hx_post=None, hx_put=None,
    hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
    hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
    hx_select_oob=None, hx_indicator=None, hx_push_url=None,
    hx_confirm=None, hx_disable=None, hx_replace_url=None,
    hx_disabled_elt=None, hx_ext=None, hx_headers=None,
    hx_history=None, hx_history_elt=None, hx_inherit=None,
    hx_params=None, hx_preserve=None, hx_prompt=None,
    hx_request=None, hx_sync=None, hx_validate=None, **kwargs)
```

An HTML partial containing a [Title](#), and [H1](#), and any provided children

Socials

[source](#)

```
Socials (title, site_name, description, image, url=None, w=1200, h=630,
    twitter_site=None, creator=None, card='summary')
```

OG and Twitter social card headers

Favicon

[source](#)

```
Favicon (light_icon, dark_icon)
```

Light and dark favicon headers

clear


[source](#)

```
clear (id)
```

with_sid

[source](#)

```
with_sid (app, dest, path='/')
```

 Report an issue

[Source](#) > Javascript examples

Javascript examples

Basic external Javascript lib wrappers

To expedite fast development, FastHTML comes with several built-in Javascript and formatting components. These are largely provided to demonstrate FastHTML JS patterns. There's far too many JS libs for FastHTML to wrap them all, and as shown here the code to add FastHTML support is very simple anyway.

light_media

[source](#)

```
light_media (css:str)
```

Render light media for day mode views

Type		Details
css	str	CSS to be included in the light media query

```
light_media('.body {color: green;})
```

```
<style>@media (prefers-color-scheme: light) {.body {color: green;}}</style>
```

dark_media

[source](#)

```
dark_media (css:str)
```

Render dark media for night mode views

Type		Details
css	str	CSS to be included in the dark media query

```
dark_media('.body {color: white;})
```

```
<style>@media (prefers-color-scheme: dark) {.body {color: white;}}</style>
```

MarkdownJS

[source](#)

MarkdownJS (sel='.marked')

Implements browser-based markdown rendering.

Type		Default	Details
sel	str	.marked	CSS selector for markdown elements

Usage example [here](#).

```
__file__ = '../..../fasthtml/katex.js'
```

KatexMarkdownJS

[source](#)

KatexMarkdownJS (sel='.marked', inline_delim='\$', display_delim='\$\$',
math_envs=None)

Type		Default	Details
sel	str	.marked	CSS selector for markdown elements
inline_delim	str	\$	Delimiter for inline math
display_delim	str	\$\$	Delimiter for long math
math_envs	NoneType	None	List of environments to render as display math

KatexMarkdown usage example:

```
longexample = r"""
Long example:


$$\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}$$



$$\nabla \times \vec{\mathbf{B}} = -\frac{1}{c} \frac{\partial \vec{\mathbf{E}}}{\partial t}$$


$$\nabla \cdot \vec{\mathbf{E}} = \frac{4\pi}{c} \rho$$


$$\nabla \times \vec{\mathbf{E}} = -\frac{1}{c} \frac{\partial \vec{\mathbf{B}}}{\partial t}$$


$$\nabla \cdot \vec{\mathbf{B}} = 0$$



$$\end{array}$$

"""

app, rt = fast_app(hdrs=[KatexMarkdownJS()])

@rt('/')
def get():
```

```
return Titled("Katex Examples",
    # Assigning 'marked' class to components renders content as markdown
    P(cls='marked')("Inline example:  $\sqrt{3x-1}+(1+x)^2$ "),
    Div(cls='marked')(longexample)
)
```

HighlightJS

[source](#)

```
HighlightJS (sel='pre code', langs:str|list|tuple='python', light='atom-
one-light', dark='atom-one-dark')
```

Implements browser-based syntax highlighting. Usage example [here](#).

Type		Default	Details
sel	str	pre code	CSS selector for code elements. Default is industry standard, be careful before adjusting it
langs	str list tuple	python	Language(s) to highlight
light	str	atom-one-light	Light theme
dark	str	atom-one-dark	Dark theme

SortableJS

[source](#)

```
SortableJS (sel='.sortable', ghost_class='blue-background-class')
```

Type		Default	Details
sel	str	.sortable	CSS selector for sortable elements
ghost_class	str	blue-background-class	When an element is being dragged, this is the class used to distinguish it from the rest

[Source](#) > SVG

SVG

Simple SVG FT elements

```
from nbdev.showdoc import show_doc
```

You can create SVGs directly from strings, for instance (as always, use `NotStr` or `Safe` to tell FastHTML to not escape the text):

```
svg = '<svg width="50" height="50"><circle cx="20" cy="20" r="15" fill="red"></circle>  
show(NotStr(svg))
```



You can also use libraries such as [fa6-icons](#).

To create and modify SVGs using a Python API, use the FT elements in `fasthtml.svg`, discussed below.

Note: `fasthtml.common` does NOT automatically export SVG elements. To get access to them, you need to import `fasthtml.svg` like so

```
from fasthtml.svg import *
```

Svg

[source](#)

```
Svg(*args, viewBox=None, h=None, w=None, height=None, width=None,  
    xmlns='http://www.w3.org/2000/svg', **kwargs)
```

An SVG tag; `xmlns` is added automatically, and `viewBox` defaults to `height` and `width` if not provided

To create your own SVGs, use `SVG`. It will automatically set the `viewBox` from `height` and `width` if not provided.

All of our shapes will have some convenient kwargs added by using [ft_svg](#):

ft_svg

[source](#)

```
ft_svg(tag:str, *c, transform=None, opacity=None, clip=None, mask=None,  
       filter=None, vector_effect=None, pointer_events=None,  
       target_id=None, hx_vals=None, id=None, cls=None, title=None,
```



```

style=None, accesskey=None, contenteditable=None, dir=None,
draggable=None, enterkeyhint=None, hidden=None, inert=None,
inputmode=None, lang=None, popover=None, spellcheck=None,
tabindex=None, translate=None, hx_get=None, hx_post=None,
hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
hx_target=None, hx_swap=None, hx_swap_oob=None, hx_include=None,
hx_select=None, hx_select_oob=None, hx_indicator=None,
hx_push_url=None, hx_confirm=None, hx_disable=None,
hx_replace_url=None, hx_disabled_elt=None, hx_ext=None,
hx_headers=None, hx_history=None, hx_history_elt=None,
hx_inherit=None, hx_params=None, hx_preserve=None,
hx_prompt=None, hx_request=None, hx_sync=None, hx_validate=None)

```

Create a standard *FT* element with some SVG-specific attrs

Basic shapes

We'll define a simple function to display SVG shapes in this notebook:

```
def demo(el, h=50, w=50): return show(Svg(h=h,w=w)(el))
```

Rect

[source](#)

```

Rect (width, height, x=0, y=0, fill=None, stroke=None, stroke_width=None,
rx=None, ry=None, transform=None, opacity=None, clip=None,
mask=None, filter=None, vector_effect=None, pointer_events=None,
target_id=None, hx_vals=None, id=None, cls=None, title=None,
style=None, accesskey=None, contenteditable=None, dir=None,
draggable=None, enterkeyhint=None, hidden=None, inert=None,
inputmode=None, lang=None, popover=None, spellcheck=None,
tabindex=None, translate=None, hx_get=None, hx_post=None,
hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None,
hx_target=None, hx_swap=None, hx_swap_oob=None, hx_include=None,
hx_select=None, hx_select_oob=None, hx_indicator=None,
hx_push_url=None, hx_confirm=None, hx_disable=None,
hx_replace_url=None, hx_disabled_elt=None, hx_ext=None,
hx_headers=None, hx_history=None, hx_history_elt=None,
hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None,
hx_request=None, hx_sync=None, hx_validate=None)

```

A standard SVG *rect* element

All our shapes just create regular *FT* elements. The only extra functionality provided by most of them is to add additional defined kwargs to improve auto-complete in IDEs and notebooks, and re-order parameters so that positional args can also be used to save a bit of typing, e.g:

```
demo(Rect(30, 30, fill='blue', rx=8, ry=8))
```



[source](#)

Circle

```
Circle (r, cx=0, cy=0, fill=None, stroke=None, stroke_width=None,
        transform=None, opacity=None, clip=None, mask=None, filter=None,
        vector_effect=None, pointer_events=None, target_id=None,
        hx_vals=None, id=None, cls=None, title=None, style=None,
        accesskey=None, contenteditable=None, dir=None, draggable=None,
        enterkeyhint=None, hidden=None, inert=None, inputmode=None,
        lang=None, popover=None, spellcheck=None, tabindex=None,
        translate=None, hx_get=None, hx_post=None, hx_put=None,
        hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
        hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
        hx_select_oob=None, hx_indicator=None, hx_push_url=None,
        hx_confirm=None, hx_disable=None, hx_replace_url=None,
        hx_disabled_elt=None, hx_ext=None, hx_headers=None,
        hx_history=None, hx_history_elt=None, hx_inherit=None,
        hx_params=None, hx_preserve=None, hx_prompt=None,
        hx_request=None, hx_sync=None, hx_validate=None)
```

A standard SVG *circle* element

```
demo(Circle(20, 25, 25, stroke='red', stroke_width=3))
```



Ellipse

[source](#)

```
Ellipse (rx, ry, cx=0, cy=0, fill=None, stroke=None, stroke_width=None,
         transform=None, opacity=None, clip=None, mask=None, filter=None,
         vector_effect=None, pointer_events=None, target_id=None,
         hx_vals=None, id=None, cls=None, title=None, style=None,
         accesskey=None, contenteditable=None, dir=None, draggable=None,
         enterkeyhint=None, hidden=None, inert=None, inputmode=None,
         lang=None, popover=None, spellcheck=None, tabindex=None,
         translate=None, hx_get=None, hx_post=None, hx_put=None,
         hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
         hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
         hx_select_oob=None, hx_indicator=None, hx_push_url=None,
         hx_confirm=None, hx_disable=None, hx_replace_url=None,
         hx_disabled_elt=None, hx_ext=None, hx_headers=None,
         hx_history=None, hx_history_elt=None, hx_inherit=None,
         hx_params=None, hx_preserve=None, hx_prompt=None,
         hx_request=None, hx_sync=None, hx_validate=None)
```

A standard SVG *ellipse* element

```
demo(Ellipse(20, 10, 25, 25))
```



transformd

[source](#)

```
transformd (translate=None, scale=None, rotate=None, skewX=None,
            skewY=None, matrix=None)
```

Create an SVG *transform* kwarg dict

```
rot = transformd(rotate=(45, 25, 25))
rot
```

```
{'transform': 'rotate(45,25,25)'}
```

```
demo(Ellipse(20, 10, 25, 25, **rot))
```



Line

[source](#)

```
Line (x1, y1, x2=0, y2=0, stroke='black', w=None, stroke_width=1,
      transform=None, opacity=None, clip=None, mask=None, filter=None,
      vector_effect=None, pointer_events=None, target_id=None,
      hx_vals=None, id=None, cls=None, title=None, style=None,
      accesskey=None, contenteditable=None, dir=None, draggable=None,
      enterkeyhint=None, hidden=None, inert=None, inputmode=None,
      lang=None, popover=None, spellcheck=None, tabindex=None,
      translate=None, hx_get=None, hx_post=None, hx_put=None,
      hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
      hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
      hx_select_oob=None, hx_indicator=None, hx_push_url=None,
      hx_confirm=None, hx_disable=None, hx_replace_url=None,
      hx_disabled_elt=None, hx_ext=None, hx_headers=None,
      hx_history=None, hx_history_elt=None, hx_inherit=None,
      hx_params=None, hx_preserve=None, hx_prompt=None, hx_request=None,
      hx_sync=None, hx_validate=None)
```

A standard SVG *line* element

```
demo(Line(20, 30, w=3))
```



Polyline

[source](#)

```
Polyline (*args, points=None, fill=None, stroke=None, stroke_width=None,
          transform=None, opacity=None, clip=None, mask=None, filter=None,
          vector_effect=None, pointer_events=None, target_id=None, hx_vals=None, id=None, cls=None, title=None,
          style=None, accesskey=None, contenteditable=None, dir=None, draggable=None, enterkeyhint=None, hidden=None, inert=None,
          inputmode=None, lang=None, popover=None, spellcheck=None, tabindex=None, translate=None, hx_get=None, hx_post=None,
          hx_put=None, hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None, hx_swap=None, hx_swap_oob=None,
          hx_include=None, hx_select=None, hx_select_oob=None, hx_indicator=None, hx_push_url=None, hx_confirm=None,
          hx_disable=None, hx_replace_url=None, hx_disabled_elt=None, hx_ext=None, hx_headers=None, hx_history=None,
          hx_history_elt=None, hx_inherit=None, hx_params=None, hx_preserve=None, hx_prompt=None, hx_request=None,
          hx_sync=None, hx_validate=None)
```

A standard SVG *polyline* element

```
demo(Polyline((0,0), (10,10), (20,0), (30,10), (40,0),
              fill='yellow', stroke='blue', stroke_width=2))
```



```
demo(Polyline(points='0,0 10,10 20,0 30,10 40,0', fill='purple', stroke_width=2))
```



Polygon

[source](#)

```
Polygon (*args, points=None, fill=None, stroke=None, stroke_width=None,
         transform=None, opacity=None, clip=None, mask=None, filter=None,
         vector_effect=None, pointer_events=None, target_id=None, hx_vals=None, id=None, cls=None, title=None, style=None,
         accesskey=None, contenteditable=None, dir=None, draggable=None, enterkeyhint=None, hidden=None, inert=None, inputmode=None,
         lang=None, popover=None, spellcheck=None, tabindex=None, translate=None, hx_get=None, hx_post=None, hx_put=None,
         hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None, hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
         hx_select_oob=None, hx_indicator=None, hx_push_url=None, hx_confirm=None, hx_disable=None, hx_replace_url=None,
         hx_disabled_elt=None, hx_ext=None, hx_headers=None, hx_history=None, hx_history_elt=None, hx_inherit=None,
         hx_params=None, hx_preserve=None, hx_prompt=None, hx_request=None, hx_sync=None, hx_validate=None)
```

A standard SVG *polygon* element

```
demo(Polygon((25,5), (43.3,15), (43.3,35), (25,45), (6.7,35), (6.7,15),
             fill='lightblue', stroke='navy', stroke_width=2))
```



```
demo(Polygon(points='25,5 43.3,15 43.3,35 25,45 6.7,35 6.7,15',
             fill='lightgreen', stroke='darkgreen', stroke_width=2))
```



Text

[source](#)

```
Text (*args, x=0, y=0, font_family=None, font_size=None, fill=None,
     text_anchor=None, dominant_baseline=None, font_weight=None,
     font_style=None, text_decoration=None, transform=None,
     opacity=None, clip=None, mask=None, filter=None,
     vector_effect=None, pointer_events=None, target_id=None,
     hx_vals=None, id=None, cls=None, title=None, style=None,
     accesskey=None, contenteditable=None, dir=None, draggable=None,
     enterkeyhint=None, hidden=None, inert=None, inputmode=None,
     lang=None, popover=None, spellcheck=None, tabindex=None,
     translate=None, hx_get=None, hx_post=None, hx_put=None,
     hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
     hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
     hx_select_oob=None, hx_indicator=None, hx_push_url=None,
     hx_confirm=None, hx_disable=None, hx_replace_url=None,
     hx_disabled_elt=None, hx_ext=None, hx_headers=None,
     hx_history=None, hx_history_elt=None, hx_inherit=None,
     hx_params=None, hx_preserve=None, hx_prompt=None, hx_request=None,
     hx_sync=None, hx_validate=None)
```

A standard SVG *text* element

```
demo(Text("Hello!", x=10, y=30))
```

Hello!

Paths

Paths in SVGs are more complex, so we add a small (optional) fluent interface for constructing them:

PathFT

[source](#)

```
PathFT (tag:str, cs:tuple, attrs:dict=None, void_=False, **kwargs)
```

A ‘Fast Tag’ structure, containing *tag*, *children*, and *attrs*

Path

```
Path (d='', fill=None, stroke=None, stroke_width=None, transform=None,
      opacity=None, clip=None, mask=None, filter=None,
      vector_effect=None, pointer_events=None, target_id=None,
      hx_vals=None, id=None, cls=None, title=None, style=None,
      accesskey=None, contenteditable=None, dir=None, draggable=None,
      enterkeyhint=None, hidden=None, inert=None, inputmode=None,
      lang=None, popover=None, spellcheck=None, tabindex=None,
      translate=None, hx_get=None, hx_post=None, hx_put=None,
      hx_delete=None, hx_patch=None, hx_trigger=None, hx_target=None,
      hx_swap=None, hx_swap_oob=None, hx_include=None, hx_select=None,
      hx_select_oob=None, hx_indicator=None, hx_push_url=None,
      hx_confirm=None, hx_disable=None, hx_replace_url=None,
      hx_disabled_elt=None, hx_ext=None, hx_headers=None,
      hx_history=None, hx_history_elt=None, hx_inherit=None,
      hx_params=None, hx_preserve=None, hx_prompt=None, hx_request=None,
      hx_sync=None, hx_validate=None)
```

Create a standard *path* SVG element. This is a special object

Let's create a square shape, but using [Path](#) instead of [Rect](#) :

- M(10, 10): Move to starting point (10, 10)
- L(40, 10): Line to (40, 10) - top edge
- L(40, 40): Line to (40, 40) - right edge
- L(10, 40): Line to (10, 40) - bottom edge
- Z(): Close path - connects back to start

M = Move to, L = Line to, Z = Close path

```
demo(Path(fill='none', stroke='purple', stroke_width=2
          ).M(10, 10).L(40, 10).L(40, 40).L(10, 40).Z()))
```



Using curves we can create a spiral:

```
p = (Path(fill='none', stroke='purple', stroke_width=2)
     .M(25, 25)
     .C(25, 25, 20, 20, 30, 20)
     .C(40, 20, 40, 30, 30, 30)
     .C(20, 30, 20, 15, 35, 15)
     .C(50, 15, 50, 35, 25, 35)
     .C(0, 35, 0, 10, 40, 10)
     .C(80, 10, 80, 40, 25, 40))
demo(p, 50, 100)
```



Using arcs and curves we can create a map marker icon:

```
p = (Path(fill='red')
      .M(25,45)
      .C(25,45,10,35,10,25)
      .A(15,15,0,1,1,40,25)
      .C(40,35,25,45,25,45)
      .Z())
demo(p)
```



Behind the scenes it's just creating regular SVG path `d` attr – you can pass `d` in directly if you prefer.

```
print(p.d)
```

```
M25 45 C25 45 10 35 10 25 A15 15 0 1 1 40 25 C40 35 25 45 25 45 Z
```

```
demo(Path(d='M25 45 C25 45 10 35 10 25 A15 15 0 1 1 40 25 C40 35 25 45 25 45 Z'))
```



PathFT.M

[source](#)

```
PathFT.M (x, y)
```

Move to.

PathFT.L

[source](#)

```
PathFT.L (x, y)
```

Line to.

PathFT.H

[source](#)

```
PathFT.H (x)
```

Horizontal line to.

PathFT.V

[source](#)

```
PathFT.V (y)
```

Vertical line to.

PathFT.Z

[source](#)

```
PathFT.Z ()
```

Close path.

PathFT.C

[source](#)

```
PathFT.C (x1, y1, x2, y2, x, y)
```

Cubic Bézier curve.

PathFT.S

[source](#)

```
PathFT.S (x2, y2, x, y)
```

Smooth cubic Bézier curve.

PathFT.Q

[source](#)

```
PathFT.Q (x1, y1, x, y)
```

Quadratic Bézier curve.

PathFT.T

[source](#)

```
PathFT.T (x, y)
```

Smooth quadratic Bézier curve.

PathFT.A

[source](#)

```
PathFT.A (rx, ry, x_axis_rotation, large_arc_flag, sweep_flag, x, y)
```

Elliptical Arc.

HTMX helpers

SvgOob

[source](#)

```
SvgOob (*args, **kwargs)
```


Wraps an SVG shape as required for an HTMX OOB swap

When returning an SVG shape out-of-band (OOB) in HTMX, you need to wrap it with [Svg0ob](#) to have it appear correctly. ([Svg0ob](#) is just a shortcut for `Template(Svg(...))`, which is the trick that makes SVG OOB swaps work.)

SvgInb

[source](#)

`SvgInb (*args, **kwargs)`

Wraps an SVG shape as required for an HTMX inband swap

When returning an SVG shape in-band in HTMX, either have the calling element include `hx_select='svg>*' ,` or `**svg_inb` (which are two ways of saying the same thing), or wrap the response with [SvgInb](#) to have it appear correctly. ([SvgInb](#) is just a shortcut for the tuple `(Svg(...), HtmxResponseHeaders(hx_reselect='svg>*))`, which is the trick that makes SVG in-band swaps work.)

 Report an issue