Tutorials > Web Devs Quickstart

Web Devs Quickstart

A fast introduction to FastHTML for experienced web developers.

Note

We're going to be adding more to this document, so check back frequently for updates.

Installation

```
pip install python-fasthtml
```

A Minimal Application

A minimal FastHTML application looks something like this:

main.py

```
from fasthtml.common import *                                   (1)

app, rt = fast_app()                                            (2)

@rt("/")                                                        (3)
def get():                                                      (4)
    return Titled("FastHTML", P("Let's do this!"))              (5)

serve()                                                         (6)
```

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`.

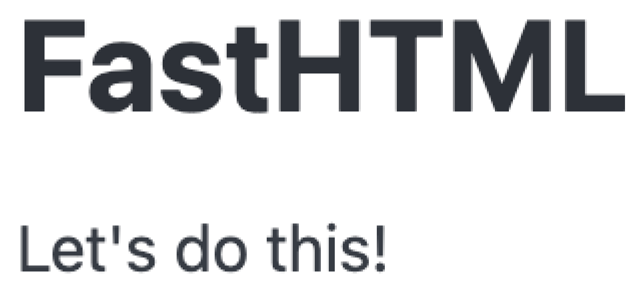
Run the code:

```
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:



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

```
from fasthtml.common import *

app, rt = fast_app()

@rt("/")                                                         1
def get():
 return Titled("FastHTML", P("Let's do this!"))

@rt("/hello")                                                    2
def get():
 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

```python
from fasthtml.common import *

app, rt = fast_app()

@rt("/{name}/{age}")
def get(name: str, age: int):
  return Titled(f"Hello {name.title()}, age {age}")

serve()
```

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

```python
 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

```python
from fasthtml.common import *

app, rt = fast_app(
    pico=False,
    hdrs=(
        Link(rel='stylesheet', href='assets/normalize.min.css', type='text/css'),
        Link(rel='stylesheet', href='assets/sakura.css', type='text/css'),
        Style("p {color: red;}")
))
```

```python

@app.get("/")
def home():
    return Titled("FastHTML",
        P("Let's do this!"),
    )

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.

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`.

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

Rendering Markdown

```python
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.

```
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>
```

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).

```python
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>
```

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

```
<div>Let&#x27;s include some HTML here: &lt;div&gt;Some HTML&lt;/div&gt;</div>
```

```python
# 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 2024-08-07 09:07:47.535449

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

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

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

```
'Cookie was set at time 09:07:47.535456'
```

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 take a little configuration plus views that use them require the `session` argument.

```
setup_toasts(app)                                                        (1)
```

```python
@rt('/toasting')
def get(session):                                                       # (2)
    # 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.

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:

```python
# 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.

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

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

Unwritten quickstart sections

- Websockets
- Tables

Report an issue