

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(): ②  
    1/0  
    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