FastMTML Static Resources

Technical Journal Entry Begins

The time has come to make what could be a 100% local web app a wee bit more so.

```
from fasthtml.common import *  # the beating heart of the app
````

...you say, Mr. Howard? Challange accepted!

Understanding Package Initialization in FastHTML Project

First, we establish our base of operations. We could look in
`/.venv/lib/python3.11/site-packages` but why when GitHub makes it all
so
pretty? So we look in
[https://github.com/AnswerDotAI/fasthtml/tree/main/fasthtml](https://g
ithub.com/AnswerDotAI/fasthtml/tree/main/fasthtml)

You'd think you'd click right into `common.py` to see what's getting
dumped into
global, but no! All ***packages*** follow the `__init__.py` rules,
which is they
execute whatever's in there first! So we look in there and find:

```python
__version__ = "0.8.1"
from .core import *
```

The Global Namespace of Files Is a Good Thing in Python

So yeah, that's a lot more being dumped into Python's file.py-level global namespace. This is not a bad thing. The nattering nabobs of namespaces plus their linting cohorts will repeatedly tell you it is a bad thing. I'm with Jeremy on this. This is one anti-pattern to embrace with gusto because it gets us nothing less the demise of the age of ReactJS—I kid you not. You're just seeing the beginning of all this play out.

HTMX Delivers Web Dev Superpowers To Python Community

But there's a price. And that price is folks like me having to do such detective work to decipher

the anti-pattern magic. It's the best kind of magic if you ask me because it's going to be your competitive advantage moat for a few years while everyone resists it. There's no taking back HTMX and FastHTML is how it's delivered like a virux into the under-powered Python community clammoring for their webdev superpowers. Shaving off the liquid template JSX moustaches in favor of a nice clean shaven Python function that looks just like an HTML() tag is it—but that one's already taken, so expect to see more inner ones like P() and Div(). That's what import * gives you.

Understanding FastHTML Import Hierarchy in core.py and common.py

Anyhow, we now know we need to look at core.py ***and*** common.py to have a handle on our import-to-global FastHTML situation. These are big files and I've listed them in blogs before so I'm not going to ruin this one by vomiting up their contents again. Instead, we're going to take a precise journey for ferreting out how to make our opinionated header imports:

```
<script
src="https://unpkg.com/htmx.org@next/dist/htmx.min.js"></script>
<script
src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-js@1.0.4/fasthtml.js"></script>
<script src="https://unpkg.com/htmx-ext-ws/ws.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>
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/@picocss/pico@latest/css/pico.min.css">
     <style>:root { --pico-font-size: 100%; }</style>
```

Rounding Up JavaScript Static Resources for Tighter Control

...a bit more under our control. Let's round them all up into a /static/ folder and make this thing run like the 100% offline-capable app it's on the verge of being (right down to the LLM if you've been following). And this is not just a gratuitous flexing of muscles for a miniscule speedup. No, this is part of getting a tighter grip on the reins of all my JavaScript and CSS which I have not been systematic enough about controlling during the development of this app, and I'm starting to get collision and race conditions.

Understanding Execution Paths in a Framework-Like

System

So, step 1: a top-down understanding of execution paths. And that starts with the "view-source" HTML page-load. And that starts with the imports that get glued together to make this system by which the rest of our code can be so clean, brief and framework-like. Client-side, these are like the Python import statements at the top of the file. It lets you know where all your off-file ingredients are coming from and is essential to knowing WTF is going on.
common.py is small enough to show again:

```
import uvicorn
from dataclasses import dataclass

from .starlette import *
from fastcore.utils import *
from fastcore.xml import *
from sqlite_minutils import Database
from fastlite import *
from .basics import *
from .pico import *
from .authmw import *
from .live_reload import *
from .toaster import *
from .js import *
from .fastapp import *
```

Convenience Wrappers Highlight Lazy Programming Approach

And this inconspicuous little line here at the bottom gives us all our convenience wrappers, and evidence of Jeremy's assertion that he's lazy. All the allowances for laziness likely can be found in fastapp.py. This one too is not too large to show again here (but just barely):

```
"""The `fast_app` convenience wrapper"""

import inspect,uvicorn
from fastcore.utils import *
from fastlite import *
from .basics import *
from .pico import *
from .starlette import *
from .live_reload import FastHTMLWithLiveReload

__all__ = ['fast_app']

def _get_tbl(dt, nm, schema):
    render = schema.pop('render', None)
    tbl = dt[nm]
```

```
    if tbl not in dt: tbl.create(**schema)
    else: tbl.create(**schema, transform=True)
    dc = tbl.dataclass()
    if render: dc.__ft__ = render
    return tbl,dc

def _app_factory(*args, **kwargs) -> FastHTML |
FastHTMLWithLiveReload:
    "Creates a FastHTML or FastHTMLWithLiveReload app instance"
    if kwargs.pop('live', False): return FastHTMLWithLiveReload(*args,
**kwargs)
    kwargs.pop('reload_attempts', None)
    kwargs.pop('reload_interval', None)
    return FastHTML(*args, **kwargs)

def fast_app(
        db_file:Optional[str]=None, # Database file name, if needed
        render:Optional[callable]=None, # Function used to render
default database class
        hdrs:Optional[tuple]=None, # Additional FT elements to add to
<HEAD>
        ftrs:Optional[tuple]=None, # Additional FT elements to add to
end of <BODY>
        tbls:Optional[dict]=None, # Experimental mapping from DB table
names to dict table definitions
        before:Optional[tuple]|Beforeware=None, # Functions to call
prior to calling handler
        middleware:Optional[tuple]=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[tuple]=None, # Passed to Starlette
        exception_handlers:Optional[dict]=None, # Passed to Starlette
        on_startup:Optional[callable]=None, # Passed to Starlette
        on_shutdown:Optional[callable]=None, # Passed to Starlette
        lifespan:Optional[callable]=None, # Passed to Starlette
        default_hdrs=True, # Include default FastHTML headers such as
HTMX script?
        pico:Optional[bool]=None, # Include PicoCSS header?
        surreal:Optional[bool]=True, # Include surreal.js/scope
headers?
        htmx:Optional[bool]=True, # Include HTMX header?
        exts:Optional[list|str]=None, # HTMX extension names to
include
        secret_key:Optional[str]=None, # Signing key for sessions
        key_fname:str='.sesskey', # Session cookie signing key file
name
```

```
        session_cookie:str='session_', # Session cookie name
        max_age:int=365*24*3600, # 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[str]=None, # Session cookie domain
        htmlkw:Optional[dict]=None, # Attrs to add to the HTML tag
        bodykw:Optional[dict]=None, # Attrs to add to the Body tag
        reload_attempts:Optional[int]=1, # Number of reload attempts
when live reloading
        reload_interval:Optional[int]=1000, # Time between reload
attempts in ms
        static_path:str=".",  # Where the static file route points to,
defaults to root dir
        body_wrap:callable=noop_body, # FT wrapper for body contents
        nb_hdrs:bool=False, # If in notebook include headers inject
headers in notebook DOM?
        **kwargs)->Any:
    "Create a FastHTML or FastHTMLWithLiveReload app."
    h = (picolink,) if pico or (pico is None and default_hdrs) else ()
    if hdrs: h += tuple(hdrs)

    app = _app_factory(hdrs=h, ftrs=ftrs, before=before,
middleware=middleware, live=live, debug=debug, routes=routes,
exception_handlers=exception_handlers,
                   on_startup=on_startup, on_shutdown=on_shutdown,
lifespan=lifespan, default_hdrs=default_hdrs, secret_key=secret_key,
                   session_cookie=session_cookie, max_age=max_age,
sess_path=sess_path, same_site=same_site,
sess_https_only=sess_https_only,
                   sess_domain=sess_domain, key_fname=key_fname,
exts=exts, surreal=surreal, htmx=htmx, htmlkw=htmlkw,
                   reload_attempts=reload_attempts,
reload_interval=reload_interval, body_wrap=body_wrap, nb_hdrs=nb_hdrs,
**(bodykw or {}))
    app.static_route_exts(static_path=static_path)
    if not db_file: return app,app.route

    db = database(db_file)
    if not tbls: tbls={}
    if kwargs:
        if isinstance(first(kwargs.values()), dict): tbls = kwargs
        else:
            kwargs['render'] = render
            tbls['items'] = kwargs
    dbtbls = [_get_tbl(db.t, k, v) for k,v in tbls.items()]
    if len(dbtbls)==1: dbtbls=dbtbls[0]
    return app,app.route,*dbtbls
```

Understanding the FastHTML Attack Against ReactJS

Drink that in, ladies and gentlemen. This is the beating heart of FastHTML. You can look other places for insights, but for a direct line into the gory vicious and deliciously understated attack against the ReactJS state of affairs, look no further than this file. Like Jeremy says, he likes to give you choice.

Choosing a Framework That Fits Your Customization Style

Just like with the Flask web microframework after which almost everything like this is patterned, you have choice. If you're a Pythonic wizard able to play whack-a-mole with some interesting framework decisions overriding the defaults of factory class instantiations, you have choice. All frameworks give you their own way of overriding opinions and customizing—or as the cool kids are saying these days, making it ***bespoke***.

FastHTML: Opinionated Conventions in Action

Anyway, when you don't have the powerful wizarding skills to override frameworks, you accept the ***opinionated conventions of the framework***. And oh it's a doozy with FastHTML. But it's my kind of doozy! But for resources loaded off of a CDN for a local app.
The way to read fast_app.py is this:

App Factory Class Inconsistent Case

Now you've got an app factory class, ho ho ho. Oddly, it's lower-cased whereas most base class template factory classes from which you instantiate objects use... use... use... Hey, Perplexity:
MikeLev.in: What is the case type called when both the initial letter and a later letter in a variable name are capitalized in programming?
Perplexity: The case type you're referring to is called "Camel Case" or more specifically "Upper Camel Case," which is also known as "Pascal Case."

Camel Case

Camel case is a naming convention where compound words or phrases are written without spaces, and each word within the name begins with a capital letter. There are two main types of camel case:

1. **Lower Camel Case (camelCase)** - The first letter of the first word is lowercase - All subsequent words start with an uppercase letter - Example: myVariableName, firstName, totalAmount

{:start="2"} 2. **Upper Camel Case (PascalCase)** - The first letter of every word, including the first one, is capitalized - Also known as Pascal Case - Example: MyClassName, FirstName, TotalAmount
Upper Camel Case (Pascal Case) is the convention you're specifically asking about, where both

the initial letter and later letters in a variable name are capitalized[1][3].

Usage in Programming

The usage of these conventions varies depending on the programming language and context:

- **Java**:
- Classes use Pascal Case (Upper Camel Case)
- Variables and methods use Lower Camel Case[1]
- **C#**:
- Classes, methods, and properties use Pascal Case
- Local variables and parameters use Lower Camel Case[3]
- **JavaScript**:
- Classes use Pascal Case
- Variables and functions typically use Lower Camel Case[2]
- **Python**:
- Classes use Pascal Case
- Functions and variables typically use snake_case (lowercase with underscores)[2]

It's important to note that while these are common conventions, they can vary between different coding standards or style guides. Consistency within a project or team is often more important than strictly adhering to a particular convention[1][2].

Citations:

[1] https://www.reddit.com/r/learnprogramming/comments/n8equt/why_does_everyone_name_their_variables_with_the/

[2] https://titlecapitalize.com/programming-case-styles/

[3] https://www.pluralsight.com/blog/software-development/programming-naming-conventions-explained

[4] https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions

[5] https://stackoverflow.com/questions/414001/variable-naming-conventions-in-java

[6] https://en.wikipedia.org/wiki/Naming_conventions_(programming)

FastAPP: Opinionated Wrapper Class Explained

Nice, thanks. So I'd expect to see FastAPP, but Jeremy goes with fast_app. Interesting choice and perhaps indicative of its baselessness, because it's not a base class at all, but a radically opinionated ***wrapper class*** with a whole bunch of defaults you can override perhaps more Pythonically, and thus easily, than one might initially imagine.

The standard pattern:

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

Flask-Inspired Factory Class Return Unpacking

...more or less following Flask's lead, but for the extra rt that comes back on the factory-class return. Already this is an opinionated decision for how many people out there, even those who claim to know Python, are comfortable with unpacking tuples on a factory class instantiation? If tuple unpacking isn't completely natural to you, dive more into the Python way.

Python Parameters Explained: Rules & Typing

Okay, so... so... so... read! Read straight down those... those... those... hey Perplexity, explain to the nice people the difference between parameters and arguments. Also explain parameter keywords and Python's parameter position rules. Oh, and throw in how the new Python mypy optional typing fits in.
Perplexity: Certainly!

Parameters vs Arguments

Parameters and arguments are often confused, but they have distinct meanings in programming:
Parameters are the variables defined in a function's declaration, while **arguments** are the actual values passed to the function when it's called.

```
def greet(name: str):  # 'name' is a parameter
    print(f"Hello, {name}!")

greet("Alice")  # "Alice" is an argument
```

Parameter Keywords and Position Rules

Python offers flexible ways to define and pass parameters:
Positional Parameters - Defined by their position in the function declaration - Must be provided in the correct order when calling the function
Keyword Parameters - Can be specified by name when calling the function - Allow arguments to be passed in any order
Default Parameters - Have a predefined value if no argument is provided
Variable-Length Parameters - *args: Allows any number of positional arguments* - *kwargs: Allows any number of keyword arguments
Here's an example demonstrating these concepts:

```
def example_function(pos1, pos2, *, kw1, kw2="default"):
    print(f"Positional: {pos1}, {pos2}")
    print(f"Keyword: {kw1}, {kw2}")

example_function(1, 2, kw1="required")
```

Parameter Position Rules

Python 3 introduced new rules for parameter ordering:

1. Positional-only parameters (separated by /)
2. Positional-or-keyword parameters
3. Keyword-only parameters (after *)

```
def complex_function(pos_only1, pos_only2, /, pos_or_kw, *, kw_only):
    pass
```

Mypy and Optional Typing

Mypy is a static type checker for Python. It supports optional typing through the Optional type hint:

```
from typing import Optional

def greet(name: Optional[str] = None):
    if name is None:
        print("Hello, stranger!")
    else:
        print(f"Hello, {name}!")
```

In this example: - name is an optional parameter - It can be either a string or None - Mypy will check for proper handling of both cases

Type Narrowing

Mypy is smart enough to perform type narrowing:

```
def process_name(name: Optional[str]):
    if name is not None:
        # Mypy knows 'name' is str here
        return name.upper()
    return "No name provided"
```

After the None check, mypy understands that name must be a string within that block, allowing string operations without additional type assertions.

By using optional typing and mypy, you can catch potential None-related errors at compile-time, improving code reliability and maintainability[1][2][4].

Citations:

[1] https://www.reddit.com/r/learnpython/comments/10rss0h/mypy_complains_about_optionalstr_even_after/
[2] https://mypy.readthedocs.io/en/stable/kinds_of_types.html
[3] https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html
[4] https://github.com/python/mypy/issues/4245
[5] https://docs.python.org/zh-cn/3/library/typing.html
[6] https://discuss.python.org/t/how-to-satisfy-mypy-optional-complaint/47760

[7] https://github.com/python/mypy/actions
[8] https://mypy-lang.org

Understanding `fast_app.py for Unprecedented Control

So there you have it. Now you know how to read fast_app.py and divine all its opinions and override them at will, right? Well fear not, you will. And it's going to be about a thousand times easier than React and put you in 1000x more control. Speaking of control, have you noticed: `default_hdrs=True`

Using Factory Class Instantiater to Remove Scripts

So, I throw caution to the wind and add that to my factory class instantiater and it has the exact intended effect of stripping out all CSS styling (for starters) and a bunch of other stuff not so immediately obvious:

```
# Unpack the returned tuple from fast_app (lots of explaining to do
here)
app, rt, (store, Store), (tasks, Task), (clients, Client) = fast_app(
    "data/data.db",
    ws_hdr=True,
    live=True,
    default_hdrs=False,
    hdrs=(
        AllTheJavaScript('.sortable'),
        Script(type='module')
    ),
    store={
        "key": str,
        "value": str,
        "pk": "key"
    },
    task={
        "id": int,
        "name": str,  # Changed from "title" to "name"
        "done": bool,
        "priority": int,
        "profile_id": int,
        "pk": "id"
    },
    client={
        "id": int,
        "name": str,
        "address": str,
```

```
        "code": str,
        "active": bool,
        "priority": int,
        "pk": "id"
    },
)
```

FastHTML Script() Element is a Powerful Semantic Map

And there you have it again. You can't see it because I don't do screenshots as it is a violation of the 80/20-rule for the reason I do these posts (my own clarity of thought vs. views). But trust me, we have a stripped-down skeletal bare bones HTML site, probably with many features like chat and sorting disabled. But not for long!
And now we look for the best approach to layering them back in. There's hand-crafting the <script> tags and injecting them somehow. But then there's this already established (in my code) way of doing it with a Script() "element" (see the blurring of the lines between Python and HTML semantics) as one of the ***arguments*** of the hdrs ***parameter***. Am I getting through to you? While I won't vomit up core.py onto this page, I will point out this choice excerpt:

```
# %% ../nbs/api/00_core.ipynb
htmxsrc   =
Script(src="https://unpkg.com/htmx.org@next/dist/htmx.min.js")
fhjsscr   =
Script(src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-js@1.0.4/
fasthtml.js")
surrsrc   =
Script(src="https://cdn.jsdelivr.net/gh/answerdotai/surreal@main/surre
al.js")
scopesrc  =
Script(src="https://cdn.jsdelivr.net/gh/gnat/css-scope-inline@main/scr
ipt.js")
viewport  = Meta(name="viewport", content="width=device-width,
initial-scale=1, viewport-fit=cover")
charset   = Meta(charset="utf-8")
```

Local Imports Streamlined Using hdrs=()

And there is our solution. We cobble together our static local imports in just such a fashion and cram it all into the parameter arguments of hdrs=() just like I managed to do with a customized version of SortableJS. And so...

```
app, rt, (store, Store), (tasks, Task), (clients, Client) = fast_app(
    "data/data.db",
    ws_hdr=True,
    live=True,
```

```
    default_hdrs=False,
    hdrs=(
        Link(rel='stylesheet', href='/static/pico.min.css'),
        Script(src='/static/htmx.min.js'),
        Script(src='/static/fasthtml.js'),
        Script(src='/static/surreal.js'),
        Script(src='/static/script.js'),
        Meta(charset='utf-8'),
        AllTheJavaScript('.sortable'),
        Script(type='module')
    ),
    store={
        "key": str,
        "value": str,
        "pk": "key"
    },
    task={
        "id": int,
        "name": str,  # Changed from "title" to "name"
        "done": bool,
        "priority": int,
        "profile_id": int,
        "pk": "id"
    },
    client={
        "id": int,
        "name": str,
        "address": str,
        "code": str,
        "active": bool,
        "priority": int,
        "pk": "id"
    },
)
```

The App Came Back, JavaScript Now Locally Hosted

Perplexity, make a little poem for "The App Came Back" in the style of The Cat Came Back.
Here's a playful poem inspired by your experience with FastHTML, echoing the style of "The Cat Came Back":

The app came back, oh what a sight,
After crashing down in the deep of night.
With imports tangled and scripts galore,
We thought we lost it, but it knocked on the door.
Chorus:

The app came back, it just wouldn’t quit,
Through bugs and errors, it wouldn’t sit.
With Python’s magic and HTML flair,
It rose from the ashes, a breath of fresh air!
It danced through the code, like a wizard at play,
With FastHTML leading the way.
“Look here,” it said, “I’m more than you know,
With every import, I’m ready to grow!”

Chorus:

The app came back, it just wouldn’t quit,
Through bugs and errors, it wouldn’t sit.
With Python’s magic and HTML flair,
It rose from the ashes, a breath of fresh air!
We wrangled the scripts and tamed every line,
From core.py to fast_app, all perfectly fine.
With each little tweak and each clever fix,
Our local web app was ready for tricks.

Chorus:

The app came back, it just wouldn’t quit,
Through bugs and errors, it wouldn’t sit.
With Python’s magic and HTML flair,
It rose from the ashes, a breath of fresh air!
So here’s to the coding and all that we do,
To frameworks and functions that help us break through.
When things go awry and you think it’s all black,
Just remember this tale: the app will come back!

Feel free to share this with your fellow coders!