FastHTML and SortableJS For Sortable Todo Lists

Technical Journal Entry Begins

Refining the Web Development Experience with Drag-and-Drop Functionality

Well, the port from yesterday hasn't technically occurred yet, however the framework is so much cleaner for "receiving" the port. I've made nice Application placeholders in navigation and with blank pages. Conceptually, it's easy to see now how I plug in new apps. And I've ton a ton of refinements on the UI to make it solid and sexy. It's really amazing what the snappiness of HTMX allows, and it's time to do drag-and-drop to get rid of "yeah-but's". I'm not a big fan of webdev, but when you do it you've got to be pretty buttoned up to modern expectations, or you're crap. Nobody wants to deal with yet another crappy interface with so many sexy new bells and whistles in tech all the time. And I think if I leave drag-and-drop sorting out of this, I'm setting users up to be disappointed given how such lists permeate everything.

Exploring FastHTML Advanced App Walk-Through for Modern Web Development

For this, we go back to the FastHTML Advanced App Walk-through
Describe the bug
SortableJS() requires presence of Script(type='module')
Minimal Reproducible Example
External scripts can be included through the fast_html() wrapper, like so:

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

...but this results in:

```
 <!doctype html>
 <html>
   <head>
<title>Botifython / Another / Home </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><scrip
t
src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-js@1.0.4/fasthtm
```

```
l.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.c
ss">
     <style>:root { --pico-font-size: 100%; }</style>

import {Sortable} from 'https://cdn.jsdelivr.net/npm/sortablejs/+esm';
proc_htmx('.sortable', Sortable.create);
<script>
    (function() {
        var socket = new
WebSocket(`ws://${window.location.host}/live-reload`);
        var maxReloadAttempts = 1;
        var reloadInterval = 1000; // time between reload attempts in
ms
        socket.onclose = function() {
            let reloadAttempts = 0;
            const intervalFn = setInterval(function(){
                window.location.reload();
                reloadAttempts++;
                if (reloadAttempts === maxReloadAttempts)
clearInterval(intervalFn);
            }, reloadInterval);
        }
    })();
</script>   </head><!-- ... -->
```

JavaScript Import Error Fixed by Module Tag Addition

...which if you look close you will see that the JavaScript import is not wrapped in a <script type="module"> tag resulting in <head>-leakage. In experimenting, I find that this will fix it:

```
app, rt = fast_app(
    hdrs=(SortableJS(), Script(type="module"))
)
```

SortableJS Script Module Tag Wrapped Correctly Despite Empty Content

...because even though it ends up with an empty script module tag, it also wraps the SortableJS one properly:

```
 <!doctype html>
 <html>
   <head>
<title>Botifython / Another / Home </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><scrip
t
src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-js@1.0.4/fasthtm
l.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.c
ss">
     <style>:root { --pico-font-size: 100%; }</style>
<script type="module">
import {Sortable} from 'https://cdn.jsdelivr.net/npm/sortablejs/+esm';
proc_htmx('.sortable', Sortable.create);
</script><script type="module"></script><script>
    (function() {
        var socket = new
WebSocket(`ws://${window.location.host}/live-reload`);
        var maxReloadAttempts = 1;
        var reloadInterval = 1000; // time between reload attempts in
ms
        socket.onclose = function() {
            let reloadAttempts = 0;
            const intervalFn = setInterval(function(){
                window.location.reload();
                reloadAttempts++;
                if (reloadAttempts === maxReloadAttempts)
clearInterval(intervalFn);
            }, reloadInterval);
        }
    })();
</script>   </head><!-- ... -->
```

It's a hack.

Expected behavior

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

```
)
```

...should result in:

```
<script type="module">
import {Sortable} from 'https://cdn.jsdelivr.net/npm/sortablejs/+esm';
proc_htmx('.sortable', Sortable.create);
</script>
```

Environment Information
- fasthtml version: 0.6.9
Confirmation
Please confirm the following:
- [✅] I have read the FAQ (https://docs.fastht.ml/explains/faq.html)
- [👨‍🔬] I have provided a minimal reproducible example
- [💯] I have included the versions of fastlite, fastcore, and fasthtml
- [🔥] I understand that this is a volunteer open source project with no commercial support.

Okay, so I submitted that on GitHub. I'm glad I found a fix. Keep in mind you can always look at the code directly. How JavaScript is included in FastHTML

Todo List Stage Set for Drag-Sortable Priorities

I have as you can see set the stage for a drag-sortable Todo list. The priority int field is already even in the table.
When using SortableJS with MiniDataAPI, updating multiple records' sortorder in a single operation isn't directly supported. However, we can implement an efficient solution that minimizes database calls and maintains data integrity. Here's a recommended approach:

Batch Update Function

Create a custom function to handle batch updates:

```
def batch_update_sortorder(table, id_order_pairs):
    with table.db.transaction():  # Assuming transaction support
        for item_id, new_order in id_order_pairs:
            table.update(id=item_id, sortorder=new_order)
```

This function:
1. Uses a transaction to ensure all updates are atomic.
2. Iterates through the new order, updating each item individually.

Client-Side Optimization

To reduce the number of updates:
1. **Track only changed positions**: On the client-side, compare the old and new positions of

items after a drag operation.

2. **Send minimal data**: Only send the IDs and new positions of items that actually changed.

```
sorter.on('sort', function(evt) {
    let changes = [];
    sorter.el.children.forEach((item, index) => {
        if (parseInt(item.dataset.sortorder) !== index) {
            changes.push({id: item.dataset.id, sortorder: index});
        }
    });

    if (changes.length > 0) {
        sendToServer(changes);
    }
});
```

Server-Side Implementation

On the server, process the changes:

```
@app.route('/update_order', methods=['POST'])
def update_order():
    changes = request.json['changes']
    batch_update_sortorder(todos, [(change['id'], change['sortorder'])
for change in changes])
    return jsonify(success=True)
```

Optimizations

1. **Incremental Updates**: For large lists, update only the range of affected items:

```
def incremental_update(table, moved_item_id, old_position,
new_position):
    with table.db.transaction():
        if new_position < old_position:
            # Moving up
            items = table(f"sortorder >= {new_position} AND sortorder
< {old_position}",
                          order_by='sortorder')
            for

 i, item in enumerate(items, start=new_position):
                table.update(id=item.id, sortorder=i)
        else:
            # Moving down
            items = table(f"sortorder > {old_position} AND sortorder
```

```
<= {new_position}",
                            order_by='sortorder')
            for i, item in enumerate(items, start=old_position+1):
                table.update(id=item.id, sortorder=i-1)

        table.update(id=moved_item_id, sortorder=new_position)
```

{:start="2"} 2. **Periodic Reordering**: Occasionally reorder all items to prevent gaps:

```
def reorder_all(table):
    items = table(order_by='sortorder')
    with table.db.transaction():
        for i, item in enumerate(items):
            table.update(id=item.id, sortorder=i)
```

By implementing these strategies, you can efficiently handle SortableJS reordering with MiniDataAPI, minimizing database operations while maintaining data consistency. This approach balances the limitations of MiniDataAPI with the need for efficient updates in a drag-and-drop interface.

Drag-and-Drop Event Already Underway With Scriptable Tag Enabled

MikeLev.in: The drag-and-drop event is already occurring. I already have the scriptable tag in place:

```
 <!doctype html>
 <html>
   <head>
<title>Botifython / First Profile3 / Todo </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><scrip
t
src="https://cdn.jsdelivr.net/gh/answerdotai/fasthtml-js@1.0.4/fasthtm
l.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.c
ss">
```

```
     <style>:root { --pico-font-size: 100%; }</style>
<script type="module">
import {Sortable} from 'https://cdn.jsdelivr.net/npm/sortablejs/+esm';
proc_htmx('.sortable', Sortable.create);
</script><script type="module"></script><script>
    (function() {
        var socket = new
WebSocket(`ws://${window.location.host}/live-reload`);
        var maxReloadAttempts = 1;
        var reloadInterval = 1000; // time between reload attempts in
ms
        socket.onclose = function() {
            let reloadAttempts = 0;
            const intervalFn = setInterval(function(){
                window.location.reload();
                reloadAttempts++;
                if (reloadAttempts === maxReloadAttempts)
clearInterval(intervalFn);
            }, reloadInterval);
        }
    })();
```

Client-Side JavaScript Required for Full HTMX Functionality

Revise your instructions knowing I need the client-side JavaScript to get or post back to an HTMX endpoint. Essentially simplify the instructions based on how far I got already. Drag and drop is working! But it's only superficial. I need to make the rubber hit the road. Or maybe the new sort order hit the MiniDataAPI spec todos object.

Breaking Free From API Constraints After Epic Adventure

Fast-forward to end of an epic adventure, and I am free from API-constraints when it's just passing stuff through. I can pass stuff through too!

```
app, rt, (store, Store), (todos, Todo), (profiles, Profile) =
fast_app(
    # ...
    hdrs=(
        SortableJSWithUpdate('.sortable',
update_url='/update_todo_order'),
        Script(type='module')
    ),
    # ...
```

```
)
```

Forcing Module Scripts in HTML Head Element Works

And so, I'm forcing it to include the ability to use module scripts, or else even the below stuff that I'm spewing in as raw JavaScript would leak out of the head element. This works...

```
def SortableJSWithUpdate(
    sel='.sortable',
    ghost_class='blue-background-class',
    update_url='/update_todo_order'
):
    src = f"""
import {{Sortable}} from
'https://cdn.jsdelivr.net/npm/sortablejs/+esm';

document.addEventListener('DOMContentLoaded', (event) => {{
    console.log('SortableJSWithUpdate script is running!');
    const el = document.querySelector('{sel}');
    if (el) {{
        new Sortable(el, {{
            animation: 150,
            ghost_class: '{ghost_class}',
            onEnd: function (evt) {{
                console.log('Drag ended!', evt);
                let items = Array.from(el.children).map((item, index)
=> ({{
                    id: item.dataset.id,
                    priority: index
                }}));
                console.log('New order:', items);

                let updateUrl = el.id === 'profile-list' ?
'/update_profile_order' : '{update_url}';
                htmx.ajax('POST', updateUrl, {{
                    target: el,  // Use the element directly instead
of a selector
                    swap: 'none',
                    values: {{
                        items: JSON.stringify(items)
                    }}
                }});
            }}
        }});
    }} else {{
        console.error('Sortable element not found:', '{sel}');
```

```
    }}
}});
"""
    return Script(src, type='module')
```

Carrying Out Client-Side htmx-Ajax Calls to Server Can Be Challenging

So, you might say I got that sorted, ahaha! Well, if you think those client-side shenanigans are fun to know the id-to-priority mappings, you'll love what it takes to carry out these client-side initiated JavaScript htmx-ajax calls to the server, through API endpoints you just blend in with the webpages under FastHTML like it's no big.
Here's what the update endpoint looks like:

```
@rt('/update_todo_order', methods=['POST'])
async def update_todo_order(values: dict):
    try:
        items = json.loads(values.get('items', '[]'))
        for item in items:
            todos.update(id=int(item['id']),
priority=int(item['priority']))
        return items
    except Exception as e:
        return str(e), 500  # Return the error message and a 500
status code
```

Implementing Sortable Todo List Items with MiniDataAPI Specification

A cascading update! There's no transaction mode, so brute force and risking integrity it is, but MiniDataAPI spec only exposes so much (lowest common denominator) database capability. But people researching this hot topic of how to reorder and sort todo list items from the FastHTML tutorial and encountered SortableJS but didn't know how to implement the JavaScript front-end/Python back-end endpoint to commit the previously "fake" sort into a "real" one might find this useful.