Why I switched from HTMX to Datastar

OCTOBER 9, 2025

In 2022, David Guillot delivered an inspiring DjangoCon Europe talk, showcasing a web app that looked and felt as dynamic as a React app. Yet he and his team had done something bold. They converted it *from* React *to* HTMX, cutting their codebase by almost 70% while significantly improving its capabilities.

Since then, teams everywhere have discovered the same thing: turning a single-page app into a multi-page hypermedia app often slashes lines of code by 60% or more while improving both developer and user experience.

I saw similar results when I switched my projects from HTMX to Datastar. It was exciting to reduce my code while building real-time, multi-user applications without needing WebSockets or complex frontend state management.

The pain point that moved the needle

While preparing my FlaskCon 2025 talk, I hit a wall. I was juggling HTMX and AlpineJS to keep pieces of my UI in sync, but they fell out of step. I lost hours debugging why my component wasn't updating. Neither library communicates with the other. Since they are different libraries created by different developers, you are the one responsible for helping them work together.

Managing the dance to initialize components at various times and orchestrating events was causing me to write more code than I wanted to and spend more time than I could spare to complete tasks.

Knowing that Datastar had the capability of both libraries with a smaller download, I thought I'd give it a try. It handled it without breaking a sweat, and the resulting code was much easier to understand.

I appreciate that there's less code to download and maintain. Having a library handle all of this in under 11 KB is great for improving page load performance, especially for users on mobile devices. The less you need to download, the better off you are.

But that's just the starting point.

Better API

As I incorporated Datastar into my project at work, I began to appreciate Datastar's API. It feels significantly lighter than HTMX. I find that I need to add fewer attributes to achieve the desired results.

For example, most interactions with HTMX require you to create an attribute to define the URL to hit, what element to target with the response, and then you might need to add more to customize how HTMX behaves, like this:

```html
<a hx-target="#rebuild-bundle-status-button"
     hx-select="#rebuild-bundle-status-button"
     hx-swap="outerHTML"
     hx-trigger="click"
     hx-get="/rebuild/status-button"></a>
```

One doesn't always need all of these, but I find it common to have two or three attributes every time[1].

With Datastar, I regularly use just one attribute, like this:

```html
<a data-on-click="@get('/rebuild/status-button')"></a>
```

This gives me less to think about when I return months later and need to recall how this works.

How to update page elements

The primary difference between HTMX and Datastar is that HTMX is a front-end library that advances the HTML specification. DataStar is a server-side-driven library that aims to create high-performance, web-native, live-updating web applications.

In HTMX, you describe its behavior by adding attributes to the element that *triggers* the request, even if it updates something far away on the page. That's powerful, but it means your logic is scattered across multiple layers. Datastar flips that: the server decides what should change, keeping all your update logic in one place.

To cite an example from HTMX's documentation:

```html
<div>
   <div id="alert"></div>
    <button hx-get="/info"
                hx-select="#info-details"
                hx-swap="outerHTML"
                hx-select-oob="#alert">
            Get Info!
    </button>
</div>
```

When the button is pressed, it sends a GET request to `` `/info` ``, replaces the button with the element in the response that has the ID 'info-details', and then retrieves the element in the response with the ID 'alert', replacing the element with the same ID on the page.

This is a lot for that button element to know. To author this code, you need to know what information you're going to return from the server, which is done outside of editing the HTML. This is when HTMX loses the ”locality of behavior” I like so much.

Datastar, on the other hand, expects the server to define the behavior, and it works better.

To replicate the behavior above, you have options. The first option keeps the HTML similar to above:

```html
 <div>
    <div id="alert"></div>
    <button id="info-details"
     data-on-click="@get('/info')">
        Get Info!
    </button>
</div>
```

In this case, the server can return an HTML string with two root elements that have the same IDs as the elements they’re updating:

```html
 <p id="info-details">These are the details you are looking f
<div id="alert">Alert! This is a test.</div>
```

I love this option because it’s simple and performant.

Think at the component level

A better option would change the HTML to treat it as a component.

What is this component? It appears to be a way for the user to get more information about a specific item.

What happens when the user clicks the button? It seems like either the information appears or there is no information to appear, and instead we render an error. Either way, the component becomes static.

Maybe we could split the component into each state, first, the placeholder:

```html
<!-- info-component-placeholder.html -->
<div id="info-component">
    <button data-on-click="@get('/product/{{product.id}}/info
        Get Info!
    </button>
</div>
```

Then the server could render the information the user requests…

```html
<!-- info-component-get.html -->
<div id="info-component">
    {% if alert %}<div id="alert">{{ alert }}</div>{% endif %
    <p>{{product.additional_information}}</p>
</div>
```

…and Datastar will update the page to reflect the changes.

This particular example is a little wonky, but I hope you get the idea. Thinking at a component level is better as it prevents you from entering an invalid state or losing track of the user’s state.

…or more than one component

One of the amazing things from David Guillot’s talk is how his app updated the count of favored items even though that element was very far away from the component that changed the count.

David’s team accomplished that by having HTMX trigger a JavaScript event, which in turn triggered the remote component to issue a GET request to update itself with the most up-to-date count.

With Datastar, you can update multiple components at once, even in a synchronous function.

If we have a component that allows someone to add an item to a shopping cart:

```html
<form id="purchase-item"
      data-on-submit="@post('/add-item', {contentType: 'form'
>
```

```
  <input type=hidden name="cart-id" value="{{cart.id}}">
  <input type=hidden name="item-id" value="{{item.id}}">
  <fieldset>
    <button data-on-click="$quantity -= 1">-</button>
    <label>Quantity
      <input name=quantity type=number data-bind-quantity val
    </label>
    <button data-on-click="$quantity += 1">+</button>
  </fieldset>
  <button type=submit>Add to cart</button>
  {% if msg %}
    <p class=message>{{msg}}</p>
  {% endif %}
</form>
```

And another one that shows the current count of items in the cart:

```
 <div id="cart-count">
    <svg viewBox="0 0 10 10" xmlns="http://www.w3.org/2000/s
        <use href="#shoppingCart">
    </svg>
    {{count}}
</div>
```

Then a developer can update them both in the same request. This is one way it could look in Django:

```
 from datastar_py.consts import ElementPatchMode
from datastar_py.django import (
    DatastarResponse,
    ServerSentEventGenerator as SSE,
)

def add_item(request):
    # skipping all the important state updates
    return DatastarResponse([
        SSE.patch_elements(
            render_to_string('purchase-item.html', context=di
        ),
        SSE.patch_elements(
            render_to_string('cart-count.html', context=dict(
        ),
    ])
```

Web native

Being a part of the Datastar Discord, I appreciate that Datastar isn't just a helper script. It's a philosophy about building apps with the web's own primitives, letting the browser and the server do what they're already great at.

Where HTMX is trying to push the HTML spec forward, Datastar is more interested in promoting the adoption of web-native features, such as CSS view transitions, Server-Sent Events, and web components, where appropriate.

This has been a massive eye-opener for me, as I've long wanted to leverage each of these technologies, and now I'm seeing the benefits.

One of the biggest wins I achieved with Datastar was by refactoring a complicated AlpineJS component and extracting a simple web component that I reused in multiple places[2] .

I especially appreciate this because there are times when it's best to rely on JavaScript to accomplish a task. But it doesn't mean you have to reach for a tool like React to achieve it. Creating custom HTML elements is a great pattern to accomplish tasks with high locality of behavior and the ability to reuse them across your app.

However, Datastar provides you with even more capabilities.

Real-time updates for multi-user apps

Apps built with collaboration as a first-class feature stand out from the rest, and Datastar is up to the challenge.

To accomplish this, most HTMX developers achieve updates either by "pulling" information from the server by polling every few seconds or by writing custom WebSocket code, which increases complexity.

Datastar uses a simple web technology called Server-Sent Events (SSE) to allow the server to "push" updates to connected clients. When something changes, such as a user adding a comment or a status change, the server can immediately update browsers with minimal additional code.

You can now build live dashboards, admin panels, and collaborative tools without crafting custom JavaScript. Everything flows from the server, through HTML.

Additionally, suppose a client's connection is interrupted. In that case, the browser will automatically attempt to reconnect without requiring additional code, and it can even notify the server, "This is the last event I received." It's wonderful.

Just because you can do it doesn't mean you should

Being a part of the Datastar community on Discord has helped me appreciate the Datastar vision of making web apps. They aim to have push-based UI updates, reduce complexity, and leverage tools like web components to handle more complex situations locally. It's common for the community to help newcomers by helping them realize they're overcomplicating things.

Here are some of the tips I've picked up:

- Don't be afraid to re-render the whole component and send it down the pipe. It's easier, it probably won't affect performance too much, you get better compression ratios, and it's incredibly fast for the browser to parse HTML strings.

- The server is the state of truth and is more powerful than the browser. Let it handle the majority of the state. You probably don't need the reactive signals as much as you think you do.

- Web components are great for encapsulating logic into a custom element with high locality of behavior. A great example of this is the star field animation in the header of the Datastar website. The `` `<ds-starfield>` `` element encapsulates all the code to animate the star field and exposes three attributes to change its internal state. Datastar drives the attributes whenever the range input changes or the mouse moves over the element.

But you can still reach for the stars

But what I'm most excited about are the possibilities that Datastar enables. The community is routinely creating projects that push well beyond the limits experienced by developers using other tools.

The examples page includes a database monitoring demo that leverages Hypermedia to significantly improve the speed and memory footprint of a demo presented at a JavaScript conference.

The one million checkbox experiment was too much for the server it started on. Anders Murphy used Datastar to create one billion checkboxes on an inexpensive server.

But the one that most inspired me was a web app that displayed data from every radar station in the United States. When a blip changed on a radar, the corresponding dot in the UI would change within 100 milliseconds. This means that *over 800,000 points are being updated per second*. Additionally, the user could scrub back in time for up to an hour (with under a 700 millisecond delay). Can you imagine this as a Hypermedia app? This is what Datastar enables.

How it’s working for me today

I’m still in what I consider my discovery phase of Datastar. Replacing the standard HTMX functionality of ajaxing updates to a UI was quick and easy to implement. Now I’m learning and experimenting with different patterns to use Datastar to achieve more and more.

For decades, I’ve been interested in ways I could provide better user experiences with real-time updates, and I love that Datastar enables me to do push-based updates, even in synchronous code.

HTMX filled me with so much joy when I started using it. But I haven’t felt like I lost anything since switching to Datastar. In fact, I feel like I’ve gained so much more.

If you’ve ever felt the joy of using HTMX, I bet you’ll feel the same leap again with Datastar. It’s like discovering what the web was meant to do all along.

learn

LINKS
About | Articles | Resources

Free! Four simple steps to solid python projects.

Reduce bugs, expand capabilities, and increase your confidence by building on these four foundational items.

Get the Guide

Join the Everyday Superpowers community!

We're building a community to help all of us grow and meet other Pythonistas. Join us!

Join

Subscribe for email updates.

your email address

Subscribe