Guide to datastar with Astro

Datastar is a hypermedia framework that gives you the backend reactivity of HTMX and the frontend reactivity of Alpine.js in a single, lightweight library.

If you've been using HTMX and Alpine together, you know the dance: HTMX handles your server requests and DOM swaps, Alpine handles your client-side interactivity, and you're constantly context-switching between two different mental models.

Datastar unifies both into one cohesive system using standard `data-*` HTML attributes and Server-Sent Events (SSE).

One library to rule them all.

In this guide I'll walk you through the building blocks of Datastar, show you how it compares to the HTMX + Alpine combo you already know, and demonstrate how to build interactive web applications with Astro on the backend.

I will assume you're familiar with HTMX and Alpine.js concepts.

Learn more on the official website at https://data-star.dev and check out the [getting started guide](https://data-star.dev/guide/getting_started).

Roger Stringer · rogerstringer.com

June 15, 2026

Contents

Why Datastar?

If you've been building web apps with HTMX and Alpine.js, you already get the hypermedia approach. You understand that sending HTML over the wire is simpler than shipping a JavaScript framework to the client. You appreciate that server-rendered partials beat JSON APIs for most UI work.

But you've probably also felt the friction of running two libraries side by side.

HTMX handles your HTTP requests, your DOM swaps, your triggers and targets. Alpine handles your client-side state, your toggles, your show/hide logic, your reactive bindings. They're both excellent at what they do, but they don't talk to each other natively. You end up bridging the gap yourself — dispatching custom events, syncing state between Alpine's `x-data` and HTMX's request cycle, managing two sets of attributes with two different mental models.

Datastar eliminates that gap entirely.

It's a single ~10KB library that gives you everything HTMX does (backend-driven DOM updates via HTTP) and everything Alpine does (reactive frontend state, conditional rendering, two-way binding) in one unified system.

Instead of `hx-get` and `x-data`, you have `data-*` attributes that handle both concerns. Instead of HTMX's innerHTML swap strategy, Datastar uses DOM morphing by default — which is smarter about preserving state. Instead of Alpine's `x-show` and `x-bind`, you have `data-show` and `data-bind` that work seamlessly with the same signals your backend can patch via SSE.

The key insight is this: Datastar treats frontend reactivity and backend communication as one problem, not two. Your backend sends HTML *and* signal updates in the same SSE stream. Your frontend reacts to user input *and* server responses using the same signal system.

If you're comfortable with HTMX + Alpine, Datastar will feel immediately familiar — but simpler. One library, one mental model, one set of attributes.

Why Datastar? Because two libraries doing the job of one is one library too many.

The Core Idea

Datastar is built on two core ideas that work together:

1. **The backend drives the frontend** by patching HTML elements and signals via Server-Sent Events (SSE).
2. **The frontend is reactive** using signals — reactive variables that automatically propagate changes through the DOM.

If you know HTMX, you know the first part. Your server returns HTML, and the client puts it on the page. But where HTMX uses a request-response cycle (make a request, get HTML back, swap it in), Datastar uses SSE streams. This means your server can send *multiple* updates in a single connection — patch an element, update a signal, patch another element, all in sequence.

If you know Alpine.js, you know the second part. You declare reactive state, bind it to the DOM, and the UI updates automatically. But where Alpine scopes state to `x-data` blocks, Datastar's signals are global. Any element anywhere in the DOM can read or write any signal.

Here's what a simple interaction looks like. In HTMX + Alpine, you might have:

```
<!-- HTMX for the request -->
<button hx-get="/greeting" hx-target="#result">Say hello</button>
<div id="result"></div>

<!-- Alpine for reactivity -->
<div x-data="{ count: 0 }">
  <button @click="count++" x-text="count"></button>
</div>
```

Two separate systems, two separate concerns. In Datastar, both live together:

```
<div data-signals:count="0">
  <!-- Backend request -->
  <button data-on:click="@get('/greeting')">Say hello</button>
  <div id="result"></div>

  <!-- Frontend reactivity -->
  <button data-on:click="$count++" data-text="$count"></button>
</div>
```

Same `data-*` attribute system for everything. The `$count` signal is reactive (like Alpine), and `@get('/greeting')` sends a request to the backend that can patch elements *and* update signals (like HTMX, but more powerful).

SSE instead of request-response

This is the biggest architectural difference from HTMX. When Datastar makes a backend request, it expects an SSE stream (or simple HTML/JSON responses for basic cases). This means your server can:

```
event: datastar-patch-elements
data: elements <div id="result">Hello!</div>

event: datastar-patch-signals
data: signals {"count": 42}
```

In one response, you've updated the DOM *and* changed frontend state. With HTMX + Alpine, you'd need out-of-band swaps, custom events, or HX-Trigger headers to accomplish the same thing.

Datastar also supports simple responses — return `text/html` and it morphs elements by ID, return `application/json` and it patches signals. But SSE is where Datastar really shines, especially for streaming updates, long-running operations, and real-time features.

Installing Datastar

Installing Datastar is straightforward. The simplest approach is a `<script>` tag from a CDN:

```
<script type="module" src="https://cdn.jsdelivr.net/gh/starfederation/datastar@1.0.0-RC.7/
bundles/datastar.js"></script>
```

Notice the `type="module"` — Datastar is distributed as an ES module. This is different from HTMX which uses a classic script tag.

In an Astro project, you'd add this to your base layout component so it's available on every page:

```
---
// src/layouts/Base.astro
---
<html lang="en">
<head>
  <script type="module" src="https://cdn.jsdelivr.net/gh/starfederation/datastar@1.0.0-
RC.7/bundles/datastar.js"></script>
</head>
<body>
  <slot />
</body>
</html>
```

If you prefer to host the file yourself, download the script or create your own bundle using the bundler (https://data-star.dev/bundler), then include it from the appropriate path:

```
<script type="module" src="/js/datastar.js"></script>
```

You can also import Datastar in a JavaScript/TypeScript file:

```
// @ts-expect-error (only required for TypeScript projects)
import 'https://cdn.jsdelivr.net/gh/starfederation/datastar@1.0.0-RC.7/bundles/
datastar.js'
```

That's it. No npm install, no build step required. Just like HTMX, one script tag and you're ready to go.

IDE Support

Datastar has official extensions for both VSCode (https://marketplace.visualstudio.com/items?itemName=starfederation.datastar-vscode) and IntelliJ (https://plugins.jetbrains.com/plugin/26072-datastar-support) that provide autocompletion for all `data-*` attributes. I'd recommend installing one of these — it makes learning the API much faster.

What about HTMX?

If you're migrating from HTMX, you can run both libraries side by side during the transition. HTMX uses `hx-*` attributes and Datastar uses `data-*` attributes, so they won't conflict. This lets you migrate page by page rather than doing a big bang rewrite.

Frontend Reactivity

If you've used Alpine.js, Datastar's frontend reactivity will feel like home. But instead of `x-data`, `x-show`, `x-bind`, and `x-text`, you're using standard `data-*` attributes.

Signals

Signals are Datastar's version of Alpine's reactive data. They're reactive variables that automatically track and propagate changes. You reference them with a `$` prefix.

In Alpine:

```
<div x-data="{ name: '' }">
  <input x-model="name" />
  <p x-text="name"></p>
</div>
```

In Datastar:

```
<div data-signals:name="''">
  <input data-bind:name />
  <p data-text="$name"></p>
</div>
```

A few key differences:

- **Signals are global.** Unlike Alpine's `x-data` which scopes state to an element, Datastar signals are accessible from anywhere in the DOM. No more worrying about whether you're inside the right `x-data` scope.
- **The `$` prefix** denotes a signal in expressions. `$name` reads the signal, `$name = 'foo'` writes to it.
- **Hyphenated names become camelCase.** `data-signals:first-name` creates a signal called `$firstName`.

data-bind

`data-bind` is Datastar's equivalent of Alpine's `x-model`. It sets up two-way binding on input elements.

```
<input data-bind:email />
```

This creates a `$email` signal bound to the input's value. Change the input, the signal updates. Change the signal, the input updates.

You can also write it as:

```
<input data-bind="email" />
```

Both syntaxes work identically.

data-text

`data-text` sets the text content of an element, like Alpine's `x-text`.

```
<input data-bind:name />
<p data-text="$name"></p>
```

The value is a Datastar expression, so you can use JavaScript:

```
<p data-text="$name.toUpperCase()"></p>
<p data-text="`Hello, ${$name}!`"></p>
```

data-show

Conditionally show or hide elements, just like Alpine's `x-show`:

```
<input data-bind:query />
<div data-show="$query !== ''" style="display: none">
  Searching for: <span data-text="$query"></span>
</div>
```

Notice the `style="display: none"` on the element. This prevents a flash of content before Datastar processes the attribute. Same trick you'd use with Alpine's `x-cloak`, but simpler.

data-computed

This one doesn't have a direct Alpine equivalent (though it's similar to a getter in `x-data`). It creates a read-only signal derived from other signals:

```
<div data-signals:price="10" data-signals:quantity="1">
  <div data-computed:total="$price * $quantity"
       data-text="`Total: $${$total}`"></div>
</div>
```

The `$total` signal automatically updates whenever `$price` or `$quantity` changes.

data-class

Toggle CSS classes based on expressions, like Alpine's `:class`:

```
<button data-class:active="$isOpen">Menu</button>
```

For multiple classes:

```
<button data-class="{active: $isOpen, 'font-bold': $isImportant}">Menu</button>
```

data-attr

Bind any HTML attribute to an expression, like Alpine's `:disabled`, `:aria-hidden`, etc.:

```
<button data-attr:disabled="$loading">Submit</button>
<div data-attr:aria-hidden="!$isVisible"></div>
```

data-on

Attach event listeners, like Alpine's `@click`, `@keydown`, etc.:

```
<button data-on:click="$count++">Increment</button>
<input data-on:keydown.enter="@get('/search')" />
```

The mapping from Alpine is direct: `@click="count++"` becomes `data-on:click="$count++"`. The main difference is the `$` prefix for signals and the `@` prefix for backend actions.

Backend Actions

This is where Datastar replaces HTMX. Instead of `hx-get`, `hx-post`, `hx-put`, `hx-patch`, and `hx-delete`, Datastar uses action functions that you call from expressions.

In HTMX:

```
<button hx-get="/data" hx-target="#result">Load</button>
```

In Datastar:

```
<button data-on:click="@get('/data')">Load</button>
```

All the HTTP verbs are available:

- `@get('/url')` — GET request
- `@post('/url')` — POST request
- `@put('/url')` — PUT request
- `@patch('/url')` — PATCH request
- `@delete('/url')` — DELETE request

How it differs from HTMX

With HTMX, you describe *what* should happen using attributes: `hx-get` for the URL, `hx-target` for where to put the response, `hx-swap` for how to insert it. It's declarative.

With Datastar, the backend *tells* the frontend what to do via SSE events. There's no `hx-target` equivalent because the server's response includes the target element IDs in the HTML it sends back. The morphing engine matches elements by their `id` attribute.

This is a fundamental shift. In HTMX, the frontend decides where content goes. In Datastar, the backend decides.

Signals are sent automatically

When Datastar makes a backend request, it automatically sends all current signal values. For GET requests, signals are serialized as a JSON string in a `datastar` query parameter. For POST/PUT/PATCH/DELETE, they're sent in the request body as JSON.

In HTMX + Alpine, you'd need to manually coordinate which data to send using `hx-include`, `hx-vals`, or form data. With Datastar, the server always knows the full frontend state.

For example, if you have:

```
<div data-signals:search="''" data-signals:page="1">
  <input data-bind:search />
  <button data-on:click="@get('/results')">Search</button>
  <div id="results"></div>
</div>
```

When the button is clicked, the GET request to `/results` will include `?datastar={"search":"...","page":1}` automatically. No `hx-include` needed.

Server-side, you extract signals from the request:

```
// For GET requests
const url = new URL(request.url)
const signals = JSON.parse(url.searchParams.get('datastar') || '{}')
const { search, page } = signals

// For POST requests
const signals = await request.json()
const { search, page } = signals
```

Triggering actions from events

Since actions are just expressions, you can call them from any event:

```
<!-- On click -->
<button data-on:click="@get('/data')">Load</button>

<!-- On form submit -->
<form data-on:submit.prevent="@post('/submit')">
  <input data-bind:name />
  <button type="submit">Send</button>
</form>

<!-- On keydown -->
<input data-bind:search data-on:keydown.debounce_500ms="@get('/search')" />

<!-- On page load -->
<div data-on:load="@get('/initial-data')"></div>
```

Notice `data-on:submit.prevent` — modifiers work like Alpine's event modifiers. And `data-on:keydown.debounce_500ms` gives you built-in debouncing, which in HTMX you'd configure with `hx-trigger="keyup changed delay:500ms"`.

Patching Elements

In HTMX, when the server responds with HTML, you tell HTMX where to put it using `hx-target` and how to insert it using `hx-swap`. The default swap is `innerHTML`.

Datastar takes a different approach. The server sends HTML elements with `id` attributes, and Datastar's morphing engine automatically matches them to existing elements in the DOM and updates them in place.

The simple case: HTML responses

If your endpoint returns `text/html`, Datastar will morph each top-level element into the DOM by matching IDs.

Frontend:

```
<button data-on:click="@get('/greeting')">Say hello</button>
<div id="message">Waiting...</div>
```

Backend response (just return HTML with a matching id):

```
<div id="message">Hello, world!</div>
```

Datastar finds the element with `id="message"` in the DOM and morphs it. No `hx-target` needed — the ID *is* the target.

Morphing vs swapping

This is one of Datastar's biggest advantages over HTMX's default behavior. HTMX's `innerHTML` swap replaces the entire content of the target. Datastar's morph compares the new HTML with the existing DOM and only changes what's different.

This means:

- Input focus is preserved
- Scroll position is maintained
- CSS transitions aren't interrupted
- Signal state on child elements survives updates

HTMX has a morph extension (`hx-swap="morph"`) but it's opt-in. In Datastar, it's the default.

Multiple elements in one response

You can update multiple parts of the page in a single response by returning multiple elements with different IDs:

```
<div id="message">Hello!</div>
<div id="count">42 messages</div>
<nav id="breadcrumb">Home > Messages</nav>
```

All three elements get morphed into the DOM. In HTMX, you'd need out-of-band swaps (`hx-swap-oob="true"`) to update multiple targets. With Datastar, it just works.

SSE: The power move

For more control, return an SSE stream (`text/event-stream`). This lets you send multiple patch events in sequence:

```
event: datastar-patch-elements
data: elements <div id="message">Loading...</div>

event: datastar-patch-elements
data: elements <div id="message">Done!</div>
```

The client processes each event in order. This is perfect for showing progress states, streaming content, or updating the UI in stages.

You can also control the merge strategy per event:

```
event: datastar-patch-elements
data: merge inner
data: elements <div id="list"><li>New item</li></div>
```

Available merge strategies:

- `morph` (default) — intelligently diff and update
- `inner` — replace innerHTML (like HTMX's default)
- `outer` — replace the entire element (like `hx-swap="outerHTML"`)
- `prepend` — add before existing content (like `hx-swap="afterbegin"`)
- `append` — add after existing content (like `hx-swap="beforeend"`)
- `before` — insert before the element (like `hx-swap="beforebegin"`)
- `after` — insert after the element (like `hx-swap="afterend"`)
- `upsert-attributes` — only update attributes, not content
- `delete` — remove the element (like `hx-swap="delete"`)

Patching Signals

Here's something HTMX simply can't do: your backend can update frontend state directly.

With HTMX + Alpine, if the server needs to change a piece of client-side state (say, close a modal or update a counter), you have to get creative. Maybe you return an `HX-Trigger` header that fires a custom event, and Alpine listens for it. Or you return HTML that includes an Alpine component with the new state baked in.

With Datastar, the server just patches the signals.

JSON responses

The simplest way: return `application/json` from your endpoint, and Datastar will merge it into the existing signals.

Frontend:

```
<div data-signals:user="''" data-signals:logged-in="false">
  <button data-on:click="@get('/check-auth')">Check Auth</button>
  <div data-show="$loggedIn">
    Welcome, <span data-text="$user"></span>!
  </div>
</div>
```

Backend (return JSON):

```
{"user": "Roger", "loggedIn": true}
```

That's it. Datastar patches those signals, the `data-show` expression re-evaluates, and the welcome message appears with the user's name. No HTML was sent — just data.

SSE signal patches

For more control, use SSE:

```
event: datastar-patch-signals
data: signals {"user": "Roger", "loggedIn": true}
```

And you can combine element patches with signal patches in the same stream:

```
event: datastar-patch-elements
data: elements <div id="nav"><a href="/profile">Profile</a></div>

event: datastar-patch-signals
data: signals {"user": "Roger", "loggedIn": true}
```

One request, one response stream, and you've updated both the DOM and the frontend state. This is the unified model that makes Datastar so powerful.

Removing signals

You can remove a signal by setting its value to `null` in a patch. Datastar uses JSON Merge Patch (RFC

7396) (https://datatracker.ietf.org/doc/rfc7396/) semantics:

```
{"tempData": null}
```

This removes `$tempData` from the signal store entirely.

Nested signals

Signals can be nested using dot notation on the frontend or nested objects from the backend:

Frontend:

```
<div data-signals:form.name="''" data-signals:form.email="''">
  <input data-bind:form.name placeholder="Name" />
  <input data-bind:form.email placeholder="Email" />
</div>
```

Backend can patch nested signals:

```
{"form": {"name": "Roger", "email": "roger@example.com"}}
```

Real-world example: form submission with validation

```
<div data-signals:errors="{}" data-signals:success="false">
  <form data-on:submit.prevent="@post('/submit')">
    <input data-bind:form.name />
    <span data-show="$errors.name" data-text="$errors.name" class="text-red-500"></span>

    <input data-bind:form.email />
    <span data-show="$errors.email" data-text="$errors.email" class="text-red-500"></span>

    <button type="submit">Save</button>
  </form>
  <div data-show="$success">Saved successfully!</div>
</div>
```

On success, the server returns:

```
{"errors": {}, "success": true}
```

On validation failure:

```
{"errors": {"name": "Name is required", "email": "Invalid email"}, "success": false}
```

The UI updates automatically based on the signals. No JavaScript event handlers, no manual DOM manipulation, no coordinating between HTMX and Alpine.

Forms & Two-Way Binding

Forms are where Datastar's unified model really shines. With HTMX, forms submit via `hx-post` and send form data. With Alpine, you manage form state via `x-model`. With Datastar, `data-bind` handles the binding and `@post` handles the submission — and the server gets all signals automatically.

Basic form

```
<div data-signals:name="''" data-signals:email="''">
  <form data-on:submit.prevent="@post('/api/contact')">
    <label>
      Name
      <input data-bind:name type="text" />
    </label>
    <label>
      Email
      <input data-bind:email type="email" />
    </label>
    <button type="submit">Send</button>
  </form>
  <div id="result"></div>
</div>
```

When the form submits, Datastar sends all signals as JSON in the request body:

```
{"name": "Roger", "email": "roger@example.com"}
```

The server can respond with HTML to update the `#result` div, or JSON to update signals, or both via SSE.

Compare this with HTMX, where you'd use `hx-post` on the form and receive `formData` server-side:

```
<!-- HTMX approach -->
<form hx-post="/api/contact" hx-target="#result">
  <input name="name" type="text" />
  <input name="email" type="email" />
  <button type="submit">Send</button>
</form>
```

The Datastar version is more explicit about what data exists (signals are declared upfront) and gives you two-way binding for free.

Select, checkbox, and radio

`data-bind` works with all input types:

```
<div data-signals:color="'red'" data-signals:agree="false" data-signals:plan="'free'">
  <!-- Select -->
  <select data-bind:color>
    <option value="red">Red</option>
    <option value="blue">Blue</option>
    <option value="green">Green</option>
  </select>

  <!-- Checkbox -->
  <label>
    <input type="checkbox" data-bind:agree />
    I agree to the terms
  </label>

  <!-- Radio -->
  <label>
    <input type="radio" data-bind:plan value="free" />
```

```
    Free
  </label>
  <label>
    <input type="radio" data-bind:plan value="pro" />
    Pro
  </label>

  <p data-text="`Color: ${$color}, Agreed: ${$agree}, Plan: ${$plan}`"></p>
</div>
```

Textarea

```
<div data-signals:bio="''">
  <textarea data-bind:bio></textarea>
  <div data-text="`${$bio.length} characters`"></div>
</div>
```

Dynamic form validation

Combine signals with `data-show` and `data-class` for real-time validation:

```
<div data-signals:password="''"
     data-computed:strong="$password.length >= 8">
  <input type="password" data-bind:password />
  <div data-show="$password.length > 0 && !$strong"
       class="text-red-500">
    Password must be at least 8 characters
  </div>
  <div data-show="$strong" class="text-green-500">
    Strong password
  </div>
  <button data-attr:disabled="!$strong"
          data-on:click="@post('/register')">
    Register
  </button>
</div>
```

In Alpine + HTMX, you'd need `x-data` for the validation state, `x-model` for the binding, and `hx-post` for the submission. Three concerns from two libraries. In Datastar, it's all one system.

Indicators & CSS Classes

When making backend requests, you want to show the user that something is happening. HTMX has the `htmx-indicator` CSS class that shows elements during requests. Datastar has a similar but more flexible system using the `data-indicator` attribute.

Request indicators

The `data-indicator` attribute creates a boolean signal that's `true` while a request from the element (or its children) is in flight.

```
<div data-signals:items="[]">
  <button data-on:click="@get('/items')"
          data-indicator:loading>
    Load Items
  </button>
  <span data-show="$loading">Loading...</span>
  <div id="items"></div>
</div>
```

When the button is clicked, `$loading` becomes `true` and the "Loading..." text appears. When the response completes, `$loading` goes back to `false`.

In HTMX, you'd do this:

```
<button hx-get="/items" hx-target="#items">
  Load Items
  <span class="htmx-indicator">Loading...</span>
</button>
```

The Datastar approach is more flexible because `$loading` is a signal — you can use it anywhere, not just inside the triggering element. You can disable buttons, change text, toggle classes, all based on the same indicator signal.

Using indicators with buttons

A common pattern is disabling the submit button and showing a spinner while a request is in flight:

```
<form data-on:submit.prevent="@post('/save')">
  <input data-bind:name />
  <button type="submit"
          data-indicator:saving
          data-attr:disabled="$saving">
    <span data-show="!$saving">Save</span>
    <span data-show="$saving">Saving...</span>
  </button>
</form>
```

CSS classes for loading states

You can combine indicators with `data-class` for visual feedback:

```
<div data-on:load="@get('/dashboard')"
     data-indicator:fetching
     data-class:opacity-50="$fetching"
     data-class:pointer-events-none="$fetching">
  <div id="dashboard">Loading dashboard...</div>
```

```
</div>
```

Toggling classes without requests

For pure frontend class toggling (no backend involved), use signals directly:

```
<div data-signals:menu-open="false">
  <button data-on:click="$menuOpen = !$menuOpen">
    Toggle Menu
  </button>
  <nav data-show="$menuOpen"
       data-class:slide-in="$menuOpen"
       style="display: none">
    <a href="/">Home</a>
    <a href="/about">About</a>
  </nav>
</div>
```

This is pure Alpine territory — but in Datastar, if you later need that menu state to affect a backend request, the signal is already there. No refactoring needed.

Multiple indicators

You can have different indicators for different requests on the same page:

```
<div>
  <button data-on:click="@get('/users')"
          data-indicator:loading-users>
    Load Users
  </button>
  <button data-on:click="@get('/posts')"
          data-indicator:loading-posts>
    Load Posts
  </button>

  <span data-show="$loadingUsers">Loading users...</span>
  <span data-show="$loadingPosts">Loading posts...</span>
  <span data-show="$loadingUsers || $loadingPosts">Something is loading...</span>
</div>
```

Try doing that with HTMX's `htmx-indicator` class. You can't — it's tied to the request element. Datastar's signal-based approach gives you full control.

Using Datastar with Astro

Astro and Datastar are a natural fit. Astro gives you server-rendered HTML with API routes, and Datastar gives you interactivity without shipping a JavaScript framework to the client.

Here's how to set up a practical Datastar + Astro workflow.

Helper utilities

First, create a helper file to simplify creating Datastar SSE responses. Here's a utility you can drop into your project:

```
// src/lib/datastar.ts

export function createDatastarResponse(
  html: string,
  signals?: Record<string, any>
) {
  const encoder = new TextEncoder()

  const readableStream = new ReadableStream({
    start(controller) {
      const minifiedHtml = html.replace(/\s+/g, ' ').trim()
      const patchElements = `event: datastar-patch-elements\ndata: elements
${minifiedHtml}\n\n`
      controller.enqueue(encoder.encode(patchElements))

      if (signals) {
        const patchSignals = `event: datastar-patch-signals\ndata: signals
${JSON.stringify(signals)}\n\n`
        controller.enqueue(encoder.encode(patchSignals))
      }

      controller.close()
    },
  })

  return new Response(readableStream, {
    headers: {
      'Content-Type': 'text/event-stream',
      'Cache-Control': 'no-cache',
      Connection: 'keep-alive',
    },
  })
}

export function createDatastarJsonResponse(
  signals: Record<string, any>
) {
  const encoder = new TextEncoder()

  const readableStream = new ReadableStream({
    start(controller) {
      const patchSignals = `event: datastar-patch-signals\ndata: signals
${JSON.stringify(signals)}\n\n`
      controller.enqueue(encoder.encode(patchSignals))
      controller.close()
    },
  })

  return new Response(readableStream, {
    headers: {
      'Content-Type': 'text/event-stream',
      'Cache-Control': 'no-cache',
      Connection: 'keep-alive',
    },
  })
}

export function getSignalsFromRequest(
  request: Request
): Record<string, any> {
  const url = new URL(request.url)
  const signalsParam = url.searchParams.get('datastar')
  if (!signalsParam) return {}
  try {
```

```
    return JSON.parse(signalsParam)
  } catch (e) {
    return {}
  }
}

export async function readSignals(
  request: Request
): Promise<Record<string, any>> {
  if (request.method === 'GET') {
    return getSignalsFromRequest(request)
  }
  const body = await request.text()
  try {
    return JSON.parse(body)
  } catch (e) {
    return {}
  }
}
```

Example: User list with search

Here's a complete example of a searchable user list.

The page (`src/pages/users.astro`):

```
---
import Layout from '../layouts/Base.astro'
---
<Layout>
  <div data-signals:search="''">
    <input data-bind:search
           data-on:keydown.debounce_300ms="@get('/api/users')"
           placeholder="Search users..." />
    <div id="user-list">Type to search...</div>
  </div>
</Layout>
```

The API route (`src/pages/api/users.ts`):

```
import type { APIRoute } from 'astro'
import { getSignalsFromRequest, createDatastarResponse } from '@/lib/datastar'

export const GET: APIRoute = async ({ request }) => {
  const signals = getSignalsFromRequest(request)
  const { search } = signals

  // Fetch users from your data source
  const users = await getUsers(search)

  const html = `
    <div id="user-list">
      ${users.length === 0
        ? '<p>No users found</p>'
        : users.map(u => `<div class="user-card">${u.name}</div>`).join('')
      }
    </div>
  `

  return createDatastarResponse(html, {
    resultCount: users.length
  })
}
```

Notice how the API route returns both HTML (the user list) and signals (the result count) in one SSE response.

Example: Modal with dynamic content

```
---
```

```
// src/pages/projects.astro
import Layout from '../layouts/Base.astro'
---
<Layout>
  <div data-signals:modal-open="false">
    <button data-on:click="$modalOpen = true; @get('/api/project/123')">
      View Project
    </button>

    <div data-show="$modalOpen"
         class="modal-overlay"
         data-on:click="$modalOpen = false"
         style="display: none">
      <div id="modal-content"
           class="modal-body"
           data-on:click.stop>
        Loading...
      </div>
    </div>
  </div>
</Layout>
```

The API route:

```
import type { APIRoute } from 'astro'
import { createDatastarResponse } from '@/lib/datastar'

export const GET: APIRoute = async ({ params }) => {
  const project = await getProject(params.id)

  return createDatastarResponse(`
    <div id="modal-content" class="modal-body" data-on:click.stop>
      <h2>${project.name}</h2>
      <p>${project.description}</p>
      <button data-on:click="$modalOpen = false">Close</button>
    </div>
  `)
}
```

The modal opens instantly (signal toggle), content loads from the server (backend action), and closing is a signal toggle again. All in one system.

Astro partials

For simpler cases where you don't need SSE, Astro partials work great. Create a partial that returns plain HTML:

```
---
// src/pages/p/greeting.astro
export const partial = true
---
<div id="greeting">Hello from the server!</div>
```

Datastar will morph the response HTML into the DOM by matching element IDs, just like it does with SSE responses.

Comparison Cheat Sheet

Here's a quick reference for translating your HTMX + Alpine knowledge to Datastar.

HTMX !Datastar

HTMX	Datastar	Notes
`hx-get="/url"`	`data-on:click="@get('/url')"`	Actions are called from expressions
`hx-post="/url"`	`data-on:submit.prevent="@post('/url')"`	Same for all HTTP verbs
`hx-target="#id"`	*(automatic by element ID)*	Server sends HTML with matching IDs
`hx-swap="innerHTML"`	`merge inner` (SSE) or morph (default)	Morph is the default and usually best
`hx-swap="outerHTML"`	`merge outer` (SSE)	
`hx-swap-oob="true"`	*(automatic)*	Multiple elements morph by ID
`hx-trigger="click"`	`data-on:click`	
`hx-trigger="keyup changed delay:500ms"`	`data-on:keyup.debounce_500ms`	Built-in modifiers
`hx-trigger="load"`	`data-on:load`	
`hx-trigger="every 5s"`	`data-on:load.interval_5000ms`	
`hx-confirm="Sure?"`	`data-on:click="if(confirm('Sure?')) @delete('/item')"`	Use JavaScript in expressions
`hx-include="#field"`	*(automatic)*	All signals sent with every request
`hx-vals='{"key":"val"}'`	`data-signals:key="'val'"`	Signals are the data layer
`hx-indicator`	`data-indicator:loading`	Signal-based, more flexible
`HX-Redirect` header	`@get('/url')` in response	Or use `datastar-execute-script` SSE event
`HX-Trigger` header	`datastar-patch-signals` SSE event	Patch signals instead of triggering events

Alpine.js !Datastar

Alpine	Datastar	Notes
`x-data="{ foo: '' }"`	`data-signals:foo="''"`	Signals are global, not scoped
`x-model="foo"`	`data-bind:foo`	Two-way binding
`x-text="foo"`	`data-text="$foo"`	Note the `$` prefix
`x-show="isOpen"`	`data-show="$isOpen"`	
`x-bind:disabled="loading"`	`data-attr:disabled="$loading"`	
`x-bind:class="{ active: isOpen }"`	`data-class:active="$isOpen"`	
`@click="count++"`	`data-on:click="$count++"`	
`@click.prevent`	`data-on:click.prevent`	Same modifier syntax
`x-cloak`	`style="display: none"`	On elements with `data-show`
`x-init="fetch(...)"`	`data-on:load="@get('/...')"`	Backend-driven initialization

Alpine	Datastar	Notes
`$dispatch('event')`	Patch signals from backend	No need for custom events
`x-for`	Server-rendered loops	Backend generates the HTML

Key mental model shifts

1. **No targets.** In HTMX you tell the frontend where to put content. In Datastar, the backend sends elements with IDs and they morph automatically.
2. **Signals replace both Alpine state and HTMX data.** One system for frontend reactivity and backend communication.
3. **SSE instead of request-response.** The server can send multiple updates in one stream — elements, signals, or both.
4. **Global signals.** No more scoping issues. Any element can read or write any signal.
5. **The server always knows your state.** Signals are sent with every backend request automatically. No more `hx-include` or `hx-vals` coordination.
6. **Morph by default.** Input focus, scroll position, and transitions are preserved automatically. No more choosing swap strategies for common cases.

For the full Datastar reference, visit data-star.dev/reference (https://data-star.dev/reference).