HTMX HANDBOOK

FLAVIO COPES

Preface

This book aims to be an introduction to HTMX, a fantastic library that lets you create interactive Web Applications.

If you're unfamiliar with JavaScript, before reading this book I highly recommend reading my JavaScript Beginner's Handbook.

This book was published in late 2024.

Legal

Introduction

HTMX is an absolutely brilliant frontend library specialized in working with HTTP requests, sending data to a server, updating the UI based on some action, and so on.

It's not meant to be a UI library like Alpine, which is more oriented at adding interactivity to a page. HTMX is still a frontend library, but it's more data-oriented, and its goal is to fully replace a UI library like React or Angular, and instead use the full power of HTML (enhanced by HTMX) and hypermedia.

Sending HTML over the wire.

In this handbook I'll explain the basic building blocks of HTMX and I'll show you how you can use those handy primitives to create Web Applications with ease.

I will explain the basics of htmx.

Learn more on the official website at https://htmx.org/docs and read the book Hypermedia Systems at https://hypermedia.systems/book/contents/

Why htmx

As a backend developer, htmx might be the best thing you stumble upon, as it lets you create quite complex experiences in your Web Application without using JavaScript.

As a frontend developer that knows and works with JavaScript or TypeScript all day long, you might wonder why use htmx instead of React or any other client-side library that you can use to build a UI?

The thing is this: if you're building your application on top of one of those big frameworks (be it React, Vue, Angular, Svelte...) you are effectively building the application by using that particular framework (or library, whatever...) rules and conventions and patterns.

You are effectively a React developer, or Vue developer.

And guess what happens when you use those frameworks, typically? There's a TON of JavaScript running under the hood to power your application.

With htmx instead you are much closer to the Web as a platform. You stay much closer to the HTML, to the browser, and to the client-server interaction.

You use much more native, built-in, platform APIs.

You use client-side JavaScript as a "scripting language" to build to interactive user interfaces, as it was intended.

Why htmx? Why not. You might discover a way of building Web Applications that totally reasons with how you think.

The core idea of htmx

The core idea of htmx is to replace the approach commonly used by modern Web Applications these days that talk to a server using JSON (primarily) and then build the user interface client-side.

Instead of sending JSON from the server to the client, we send HTML.

Imagine you want to get some users data to embed in a list. You ask this data to a `/api/users` endpoint. It sends you this:

```
{
  "users": [
    { "name": "First user" },
```

```
    { "name": "Second user" },
    { "name": "Third user" },
  ]
}
```

Now you (the client) must render this data to the user, building the HTML (that is what the browser can interpret) client-side.

With HTMX, you'd see a list like this coming from the server, returned as an HTML partial (not a full HTML page, just some HTML tags):

```
<ul>
  <li>First user</li>
  <li>Second user</li>
</ul>
```

and this reply is automatically shown on the page by htmx, you just describe where you want to show it, and it's all automatic.

This is a simple example, but you can already see how mind-shifting htmx can be if you're used to passing JSON around, like we've been doing for years and years.

With htmx, API endpoints are more UI-aware. They're not pure isolated data-spitting HTTP routes. They become active parts of the application.

This was one big idea of htmx: JSON -> HTML.

Another big idea is that we can fully use all the HTTP verbs from HTML (which is otherwise limited to GET and POST) and also send PUT, PATCH and DELETE requests.

And, we can send those requests from any HTML element, not just forms (and links, for GET request).

Those requests can be triggered by any event we want, and we describe (using special htmx attributes) what should happen.

We'll see this soon in practice.

One important thing to note is that htmx is backend agnostic, we can use any backend that can render an HTML partial (basically, any backend).

Installing htmx

Installing htmx can be as simple as adding a `<script>` tag to an HTML page that loads htmx from a global CDN, like this:

```
<script src="<https://unpkg.com/htmx.org@1.9.9>"></script>
```

You can also save the htmx code locally on your website and load it from there. Or reference `https://unpkg.com/htmx.org` which gives you the latest version (but adds a redirect) - if you choose this route it's probably safer to use `https://unpkg.com/htmx.org@1` to stick to htmx 1.x in case 2.x is released with breaking changes.

In a modern site built for example with Astro you typically do this in a common layout component, but you can start simple and add it to a simple `index.html` page that you then load in the browser from your local filesystem.

You can also install htmx using npm to add it to your build system and use the `import` syntax to load it:

```
npm install htmx.org

//....in your code...

import 'htmx.org'
```

But I think the `<script>` tag approach is best suited for htmx.

Doing a GET request

Consider this button:

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

<div id="data"></div>
```

When you click the button, any HTML returned from the `GET /data` HTTP request will be put inside the element that matches the selector `#data` (in this example our `<div id="data"></div>`).

I'll use Astro to show a simple example, create a page with

```
<html lang='en'>
  <head>
    <script src='<https://unpkg.com/htmx.org@1>'></script>
  </head>
  <body>
    <button
```

```
      hx-get='/data'
      hx-target='#data'>
      Load fresh data
    </button>

    <div id='data'></div>
  </body>
</html>
```

Now create a `src/pages/data.astro` and in there add

```
---
export const partial = true
---

<p>test response</p>
```

Clicking the `Load fresh data` button will insert `<p>test response</p>` into the `#data` div.

Note that all is happening without us having to write a single line of JavaScript.

htmx does all the JavaScript for us. We just describe what we want it to do. And it's pretty flexible, so we can do a lot with it.

Swap

Using the `hx-swap` attribute we can tell htmx to use a specific swap "strategy".

The default is `innerHTML` which swaps the inner content of the selector.

But you can tell htmx to use another swap strategy with the `hx-swap` attribute.

For example `hx-swap='outerHTML'`, which swaps the element that matches the selector too (in our example the entire `<div id='data'></div>`, with all its content).

This is common if from the server you return content wrapped in the div, like this:

```
<div id='data'>
  <p>test response</p>
</div>
```

You can use several other strategies including not doing anything with `hx-swap='none'` (which is what you want to do if the request response shouldn't be swapped anywhere), or deleting an element with `hx-swap='delete'`

Or adding the response HTML at the top of a list included in the target with `afterbegin` (or at the bottom with `beforeend`).

Or add the response HTML before the target element with `beforebegin` (or after the target element with `afterend`).

This lets us do quite a lot of things.

One thing that I want to mention is "out of band swaps", because this unlocked several opportunities in my mind.

Basically in the response we can return one or more HTML tags with the `hx-swap-oob="true"` attribute that are swapped in different places in the page:

```
<div id="message" hx-swap-oob="true">
  Show this in #message
</div>

<p>Show me as normal</p>
```

POST request

We previously saw how to use `hx-get` to do a GET request.

POST requests are similar, but using `hx-post`:

```
<button hx-post="/data"
        hx-swap="innerHTML"
        hx-target="#data">
  Load fresh data
</button>

<div id="data"></div>
```

If `/data` returns the same data as the GET request, this works in the same way.

But conceptually POST requests are used to send data to the endpoint.

This is done using forms.

If the element issuing the POST is inside a form, or *is* a form, all the input fields are sent to the endpoint as form data.

You can configure this behavior by filtering out some fields using `hx-params` and including other input fields using `hx-include`.

Typically you have a form like this:

```
<form
  hx-post="/projects"
  hx-target="#result">
  <input name="name" />
  <button type="submit">Add</button>
</form>
```

This is equivalent to:

```
<form>
  <input name="name" />
  <button
    type="submit"
    hx-post="/projects"
    hx-target="#result">
    Add
  </button>
</form>
```

Both work in the same way, htmx posts to `/projects` the data of the form, which in this case means the `name` input field value.

Note that if you use validation in a form, for example setting a field as `required`, the request will not be sent if validation fails.

Server-side, for example using Astro, you can get this data using `Astro.request.formData()`:

```
---
export const partial = true

if (Astro.request.method === 'POST') {
  const formData = await Astro.request.formData()
  //this prints the form data to the console
  console.log(formData)

  //FormData { [Symbol(state)]: [ { name: 'name', value: 'my project
name' } ] }
}
---

<p>project created</p>
```

Note that you must configure Astro to be server-rendered in `astro.config.mjs`:

```
import { defineConfig } from 'astro/config'

// <https://astro.build/config>
export default defineConfig({
  **output: 'server'**
})
```

Targets

We used the `hx-target` attribute to tell htmx where to put the response of the request.

This attribute takes a CSS selector, which is what you're likely familiar to use in CSS to target an element, for example by `id` or `class`.

In our example we had `hx-target="#result"`.

That put the result of the request to the element with the `id` attribute equal to `result`

(As a reminder, the difference between `id` and `class` is that `id` attributes must be unique on a page, while `class` can be repeated multiple times, so you have to handle that case)

This is the simplest usage of `hx-target`.

You can use `hx-target="this"` to target "this element we are defining the `xh-target` attribute on".

Then we can navigate the HTML elements around or near us with:

- `closest <CSS selector>` to find the closest parent that matches the CSS selector
- `next <CSS selector>` to find the closest next element that matches the CSS selector
- `previous <CSS selector>` to find the closest previous element that matches the CSS selector
- `find <CSS selector>` to find the closest child that matches the CSS selector

For example in a project I used `hx-target="closest li"` in a list with a delete button to delete the entire `li` (list item) when I clicked the delete button to delete the item.

Loading indicator

It's common to have a loading indicator show up when we're waiting for the server response.

To do that, embed an element with class `htmx-indicator` into the element that triggers the request.

You can use a simple `loading...` text:

```
<button
  hx-get='/data'
  hx-swap='innerHTML'
  hx-target='#data'>
  Load fresh data
  <p class='htmx-indicator'>loading...</p>
</button>
```

Or an image:

```
<button
  hx-get='/data'
  hx-swap='innerHTML'
  hx-target='#data'>
  Load fresh data
  <img class='htmx-indicator' src='/spinner.gif' />
</button>
```

Confirming actions, and prompts

A handy utility in htmx is `hx-confirm` which allows you to let the user confirm an action.

For example you have a delete button to delete an item in a list, and you don't want the user to lose data if accidentally pressing the button.

So you add `hx-confirm`, adding some confirmation text:

```
<ul>
  {tasks.map(task =>
    <li>
      Task: {task.name}
      <button
        hx-confirm="Are you sure?"
        hx-target="closest li"
        hx-swap="outerHTML"
        hx-delete={`/task/${task.id}`}>
        Delete
      </button>
    </li>
  )}
</ul>
```

(Notice I used some Astro templating to print the tasks list, to show you how I used this feature).

You can also ask for some information to the user in a prompt, using `hx-prompt`:

```
<button hx-delete="/project" hx-prompt="Enter the project name to
confirm">
  Delete project
</button>
```

This information is sent in the `HX-Prompt` HTTP request header.

Triggers

Requests can be triggered in different ways.

The default is `click`, but you can fire a requests upon any browser-generated event, like `mouseenter` or `keyup` or even a specific keypress.

You do that using `hx-trigger`:

```
<button
  hx-get='/data'
  hx-swap='innerHTML'
  hx-target='#data'
  hx-trigger='mouseenter'>
  Load fresh data
</button>
```

It doesn't have to be an event.

htmx offers other ways to fire events, like polling. Use `every 5s` to fire a GET request every 5 seconds: `hx-trigger='every 5s'`

Or use `load` to load just once after 10 seconds: `hx-trigger='load delay:10s'`

Request headers

It's important to note that in any request sent using htmx we have access, server-side, to a number of HTTP headers we can use.

We've got quite a few useful ones:

- `XH-Current-URL` the URL the request comes from
- `XH-Target` the `id` of the target element
- `XH-Trigger` the `id` of the triggered element
- `XH-Trigger-Name` the `name` of the triggered element

Remember, **target** = the element we'll print the response to. **trigger** = the element that triggered the request.

We've talked about `HX-Prompt` already, if you have a prompt in the tag, you get what the user wrote in the prompt in this header.

See the full list of request headers: https://htmx.org/docs/#request-headers

Response headers

A very cool thing about htmx is that we can send a response back from the server with some special headers that then trigger client-side behavior.

Like redirecting to a specific URL determined by the server after the request is received.

This helps us create applications where client and server are tightly coupled (i.e. the decision of what happens is determined with server-side logic).

For example I used the `HX-Redirect` response header, set it to `/`, and client-side once the request was successfully completed the user was redirected to `/`.

Here's the code I used to do this:

```
if (Astro.request.method === 'DELETE') {
  await deleteProject(id) //some logic

  **return new Response(null, {
    status: 204,
    statusText: 'No Content',
    headers: {
      'HX-Redirect': '/',
    },
  })**
}
```

After doing the HTTP request, htmx automatically redirects to that URL, client-side.

Other interesting ones are:

- `HX-Push-Url` to push a new URL in the browser history (equivalent to client-side calling History API's pushState)
- `HX-Refresh` to trigger a client-side full refresh of the page after the response is received
- `HX-Trigger` to trigger a client-side event right after the response is received
- `HX-Retarget` to change the `hx-target` of the response to a different element on the page

A full list of other response headers you can set is here: https://htmx.org/docs/#response-headers

Events

I had the need to listen for a particular htmx event to happen, and add some custom JavaScript code.

htmx allows us to do this using plain DOM events.

In this example I redirect to the `/` route after the request that was triggered by the HTML element with `id` equal to `button-delete-project`:

```
<script>
  document.addEventListener('htmx:afterRequest', function (event) {
    if ((event as CustomEvent).detail.target.id === 'button-delete-
project') {
      window.location.href = '/'
    }
  })
</script>
```

The `detail` object attached to the event provides a lot of useful information.

You have a wide variety of events to listen to, including for example:

- `htmx:beforeRequest` fired before an HTTP request is issued
- `htmx:beforeSwap` fired before a DOM swap
- `htmx:timeout` fired when a request timeout occurs

You should check https://htmx.org/events/ for the full list and documentation.