Hypermedia In Action

11. Developing With htmx

This chapter covers

- The details of htmx attributes
- Events & htmx
- HTTP Requests & htmx
- Updating Other Content
- Debugging htmx Applications
- Security Considerations
- Configuring htmx

Getting Deeper Into htmx

In this chapter we are going to look more deeply into htmx. We've accomplished quite a bit with what we've learned so far, but, when you are developing Hypermedia Driven Applications, there are likely to be situations that arise that require additional functionality to address cleanly. We will go over some less commonly used attributes in htmx, as well as expand on the details of some attributes we have already used.

Additionally, we will look at the functionality that htmx offers beyond simple HTML attributes: how htmx extends standard HTTP request and responses, how htmx works with (and produces) events, and how to approach situations where there isn't a simple, single target on the page to be updated.

Finally, we will take a look at practical considerations when doing htmx development: how to debug htmx-based applications effectively, security considerations you will need to take into account when working with htmx, and how to configure the behavior of htmx.

11.1. htmx Attributes

Thus far we have, we've used about fifteen different attributes from htmx in our application. The most important ones have been:

Attribute	Use
`hx-get`, `hx-post`, etc.	To specify the AJAX request an element should make
`hx-trigger`	To specify the event that triggers a request
`hx-swap`	To specify how to swap the returned HTML content into the DOM
`hx-target`	To specify where in the DOM to swap the returned HTML content

Let's do a deep dive on two of these attributes, `hx-swap` and `hx-trigger`, because they support a large number of options that might be useful when you are creating more advanced Hypermedia Driven Applications.

11.1.1. `hx-swap`

The `hx-swap` attribute is often not included on elements that issue htmx-driven requests. This is because, for many cases, the default behavior, `innerHTML`, which swaps the inner HTML of the element, is fine. Of course, we have seen cases where we wanted to override this behavior and use `outerHTML`, for example. And, in chapter 3, we discussed some other swap options beyond these two, `beforebegin`, `afterend`, etc.

In chapter 5, we also looked at the `swap` delay modifier for `hx-swap`, which allowed us to fade some content out before it was removed from the DOM.

In addition to these, `hx-swap` also supports the following modifiers:

Modifier	Use
`settle`	Like `swap`, this allows you to apply a specific delay between when the content has been swapped into the DOM and when its attributes are "settled", that is, updated from their old values (if any) to their new values.

Modifier	Use
`show`	Allows you to specify an element that should be shown (that is, scrolled into the viewport of the browser if necessary) when a request is completed
`scroll`	Allows you to specify a scrollable element (that is, an element with scrollbars), that should be scrolled to the top or bottom when a request is completed
`focus-scroll`	Allows you to specify that htmx should scroll to the focused element when a request completes. (This defaults to false)

So, for example, if we had a button that issued a `GET` request, and we wished to scroll to the top of the `body` element when the request had completed, we would write the following HTML:

Listing 11. 1. Scrolling To The Top Of The Page

```
<button hx-get="/contacts" hx-target="#content-div"
        hx-swap="innerHTML show:body:top"> ❶
  Get Contacts
</button>
```

❶ This tells htmx to show the top of the body after the swap occurs

More details and examples can be found online at the documentation page for `hx-swap`: https://htmx.org/attributes/hx-swap/

11.1.2. `hx-trigger`

Like `hx-swap`, `hx-trigger` can often be omitted when you are using htmx, because the default behavior is typically what you want anyway. Recall the default triggering events are determined by an element's type:

- Requests on `input`, `textarea` & `select` elements are triggered by the `change` event

- Requests on `form` elements are triggered on the `submit` event
- Requests on all other elements are triggered by the `click` event

There are times, however, when you want a more elaborate trigger specification. A classic example was the active search example we implemented in Contact.app:

Listing 11. 2. The Active Search Input

```
    <input id="search" type="search" name="q" value="{{ request.args.get('q') or ''
}}"
           hx-get="/contacts"
           hx-trigger="search, keyup delay:200ms changed"/> ❶
```

❶ An elaborate trigger specification

This example took advantage of two modifiers available for the `hx-trigger` attribute:

Modifier	Use
`delay`	Allows you to specify a delay to wait before a request is issued. If the event occurs again, the first event is discarded and the timer resets. This allows you to "debounce" requests.
`changed`	Allows you to specify that a request should only be issued when the `value` property of the given element has changed

`hx-trigger` has quite a few additional modifiers. This makes sense, because events are fairly complex and we want to be able to take advantage of all the power they offer. (We will discuss events in more detail below.)

Here are the other modifiers available on `hx-trigger`:

Modifier	Use
`once`	The given event will only trigger a request once

Modifier	Use
`throttle`	Allows you to throttle events, only issuing them once every certain interval. This is different than `delay` in that the first event will trigger immediately, but any following events will not trigger until the throttle time period has elapsed
`from`	A CSS selector that allows you to pick another element to listen for events on. We will see an example of this used later in the chapter.
`target`	A CSS selector that allows you to filter events to only those that occur directly on a given element. In the DOM, events "bubble" to their parent elements, so a `click` event on a button will also trigger a `click` event on a parent `div`, all the way up to the `body` element. Sometimes you want to specify an event directly on a given element, and this attribute allows you to do that.
`consume`	If this option is set to `true`, the triggering event will be cancelled and not propagate to parent elements.

Modifier	Use
`queue`	This option allows you to specify how events are queued in htmx. By default, when htmx receives a triggering event, it will issue a request and start an event queue. If the request is still in flight when another event is received, it will queue the event and, when the request finishes, trigger a new request. By default, it only keeps the last event it receives, but you can modify that behavior using this option: for example, you can set it to `none` and ignore all triggering events that occur during a request.

Filters

The `hx-trigger` attribute allows you to specify a *filter* to events by using square brackets enclosing a JavaScript expression after the event name.

Let's say you have a complex situation where contacts should only be retrievable in certain situations, and you have a JavaScript function, `contactRetrievalEnabled()` that returns a boolean, `true` if contacts can be retrieved and `false` otherwise. You want to gate a button that issues a request to `/contacts` on this function. To do this using an event filter in htmx, you would write the following HTML:

Listing 11. 3. The Active Search Input

```
<script>
  function contactRetrievalEnabled() {
      ...
  }
</script>
<button hx-get="/contacts" hx-trigger="click[contactRetrievalEnabled()]"> ❶
  Get Contacts
</button>
```

❶ the event filter, calling `contactRetrievalEnabled()`

The button will not issue a request if `contactRetrievalEnabled()` returns false, allowing you to dynamically control when the request will be made. Common situations that call for an event trigger are:

- Only issue a request when a certain element has focus
- Only issue a request when a given form is valid
- Only issue a request when a set of inputs have specific values

Using event filters, you can use whatever logic you'd like to filter requests by htmx.

Synthetic Events

In addition to these modifiers, `hx-trigger` offers a few "synthetic" events, that is events that are not part of the regular DOM API. We have already seen `load` and `revealed` in our lazy loading and infinite scroll examples, but htmx also gives you an `intersect` event that triggers when an element intersects its parent element.

This synthetic event uses the modern Intersection Observer API, which you can read more about here: https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API

Intersection gives you much finer grained control over exactly when a request should be triggered. For example, you can specify a threshold and specify that the request should only be issued when an element is 50% visible.

The `hx-trigger` attribute certainly is the most complex on in htmx, and more details and examples can be found online its documentation page: https://htmx.org/attributes/hx-trigger/

11.1.3. Other Attributes

htmx offers many other less commonly used attributes for fine-tuning the behavior of your Hypermedia Driven Application. Here are some of the most useful ones:

Attribute	Use
hx-push-url	"Pushes" the request URL (or some other value) into the navigation bar

Attribute	Use
hx-preserve	Preserves a bit of the DOM between requests (the original content will be kept, regardless of what is returned)
hx-sync	Synchronized requests between two or more elements
hx-disable	Disables htmx behavior on this element and any children. We will discuss this more below in the security section.

Let's take a look at `hx-sync`, which allows us to synchronize AJAX requests between two or more elements. Consider a simple case where we have two buttons that both target the same element on the screen:

Listing 11. 4. Two Competing Buttons

```
<button hx-get="/contacts" hx-target="body"> ❶
  Get Contacts
</button>
<button hx-get="/settings" hx-target="body"> ❶
  Get Settings
</button>
```

This is fine and will work, but what if a user clicks the "Get Contacts" button and then the request takes a while to respond? And, in the meantime the user clicks the "Get Settings" button? In this case we would have two requests in flight at the same time.

If the `/settings` request finished first and displayed the user's setting information, they might be very surprised if they began making changes and then, suddenly, the `/contacts` request finished and replaced the entire body with the contacts instead!

To deal with this situation, we might consider using an `hx-indicator` to alert the user that something is going on, making it less likely that they click the second button. But if we really want to guarantee that there is only one request at a time issued between these two buttons, the right thing to do is to use the `hx-sync` attribute. Let's enclose both buttons in a `div` and eliminate the redundant `hx-target` specification by hoisting the attribute up to

that `div`. We can then use `hx-sync` on that div to coordinate requests between the two buttons.

Here is our updated code:

Listing 11. 5. Syncing Two Buttons

```
<div hx-target="body"  ❶
     hx-sync="this">  ❷
    <button hx-get="/contacts"> ❶
      Get Contacts
    </button>
    <button hx-get="/settings"> ❶
      Get Settings
    </button>
</div>
```

❶ Hoist the duplicate `hx-target` attributes to the parent `div`

❷ Synchronize on the parent `div`

By placing the `hx-sync` attribute on the `div` with the value `this`, we are saying "Synchronize all htmx requests that occur within this `div` element with one another." This means that if one button already has a request in flight, other buttons within the `div` will not issue requests until that has finished.

The `hx-sync` attribute supports a few different strategies that allow you to, for example, replace an existing request in flight, or queue requests with a particular queuing strategy. You can find complete documentation, as well as examples, at the documentation page for `hx-sync`: https://htmx.org/attributes/hx-sync/

As you can see, htmx offers a lot of attribute-driven functionality for more advanced Hypermedia Driven Applications. A complete reference for all htmx attributes can be found at https://htmx.org/reference/#attributes

11.2. Events

We have been working with events in htmx primary via the `hx-trigger` attribute. This has proven to be a powerful mechanism for driving our application using declarative, HTML-friendly syntax. However, there is more to events and htmx than just `hx-trigger`.

11.2.1. htmx-generated Events

It turns out that, in addition to making it easy to *respond* to events, htmx also *emits* many useful events. You can use these events to add more functionality to your application, either via htmx itself, or by way of scripting.

Here are some of the most commonly used events in htmx:

Event	Description
`htmx:load`	Triggered when new content is loaded into the DOM by htmx
`htmx:configRequest`	Triggered before a request is issued, allowing you to programmatically configure the request (or cancel it entirely)
`htmx:afterRequest`	Triggered after a request has responded
`htmx:abort`	A custom event that can be sent to an htmx-powered element to abort an open request

We have already seen how to use the `htmx:load` event, using the `htmx.onLoad()` API in the sortable.js example, which is probably one of the most common uses of events.

11.2.2. Using The `htmx:configRequest` Event

Let's take a look at how you might use the `htmx:configRequest` event to configure an HTTP request. Consider the following scenario: our server-side team has decided that they want you to include a token for extra validation on every request. The token is going to be stored in `localStorage` in the browser, in the slot `special-token`. The server-side team wants you to include this special token on every request made by htmx, as the `X-SPECIAL-TOKEN` header.

How could you achieve this? One way would be to catch the `htmx:configRequest` event and update the `detail.headers` object with this token from `localStorage`.

In VanillaJS, it would look something like this:

Listing 11. 6. Adding the `X-SPECIAL-TOKEN` Header

```
document.body.addEventListener("htmx:configRequest", function(configEvent){
    configEvent.detail.headers['X-SPECIAL-TOKEN'] = localStorage['special-token']; ❶
})
```

❶ retrieve the value from local storage and set it into a header

As you can see, we add a new value to the `headers` property of the event's detail. After the event handler executes, the `headers` property is read by htmx and used to construct the headers for an AJAX request. So, with this bit of JavaScript code, we have added a new custom header to every AJAX request that htmx makes. Slick!

You can also update the `parameters` property to change the parameters submitted by the request, change the target of the request, and so on. Full documentation for the `htmx:configRequest` event can be found here: https://htmx.org/events/#htmx:configRequest

11.2.3. Canceling a Request using `htmx:abort`

So we can listen for many useful events from htmx, and we can respond to events using `hx-trigger`. What else can we do with events? It turns out that htmx itself listens for one special event, `htmx:abort`. When htmx receives this event on an element that has a request in flight, it will abort the request.

Consider a situation where we have a potentially long-running request to `/contacts`, and we want to offer a way for the users to cancel the request. What we want is a button that issues the request, driven by htmx, of course, and then another button that will send an `htmx:abort` event to the first one.

Here is what the code might look like:

Listing 11. 7. A Button With An Abort

```
<button id="contacts-btn" hx-get="/contacts" hx-target="body"> ❶
  Get Contacts
</button>
<button onclick="document.getElementById('contacts-btn').dispatchEvent(new
Event('htmx:abort'))"> ❷
  Cancel
</button>
```

❶ A normal htmx-driven `GET` request to `/contacts`

❷ JavaScript to look up the button and send it an `htxm:abort` event

So now, if a user clicks on the "Get Contacts" button and the request takes a while, they can click on the "Cancel" button and end the request. Of course, in a more sophisticated user interface, you may want to disable the "Cancel" button unless an HTTP request is in flight, but that would be a pain to implement in pure JavaScript. Thankfully it isn't too bad to implement in hyperscript, so let's take a look at what that would look like:

Listing 11. 8. A hyperscript-Powered Button With An Abort

```
<button id="contacts-btn" hx-get="/contacts" hx-target="body">
  Get Contacts
</button>
<button _="on click send htmx:abort to #contacts-btn
           on htmx:beforeRequest from #contacts-btn remove @disabled from me
           on htmx:afterRequest from #contacts-btn add @disabled to me">
  Cancel
</button>
```

Now we have a "Cancel" button that is disabled only when a request from the `contacts-btn` button is in flight. And we are taking advantage of htmx-generated and handled events, as well as the event-friendly syntax of hyperscript, to make it happen. Not bad!

11.2.4. Server Generated Events

We are going to talk more about the various ways that htmx enhances regular HTTP requests and responses in the next section, but, since it involves events, we are going to discuss one HTTP Response header that htmx supports: `HX-Trigger`. We have discussed before how HTTP requests and responses support *headers*, name-value pairs that contain

metadata about a given request or response. We took advantage of the `HX-Trigger` request header, which includes the id of the element that triggered a given request.

It turns out that there is a *response* header, also named `HX-Trigger` in that htmx supports. This response header allows you to trigger an event on the element that submitted an AJAX request. This turns out to be a powerful way to coordinate elements in the DOM in a decoupled manner.

To see how this might work, lets consider the following situation: we have a button that grabs new contacts from some remote system on the server. We will ignore the details of the server side implementation, but we know that if we issue a `POST` to the `/integrations/1` path, it will trigger a synchronization with the system.

Now, this synchronization may or may not result in new contacts being created. In the case where new contacts *are* created, we want to refresh our contacts table. In the case where no contacts are created, we don't want to refresh the table.

How could we implement this using the `HX-Trigger` response header? Well, we could conditionally add an `HX-Trigger` response header with the value `contacts-updated`, which would trigger the `contacts-updated` event on the button that made the AJAX request to `/integrations/1`. And we can then take advantage of the `from:` modifier of the `hx-trigger` attribute to listen for that event! Now we can effectively trigger htmx request from the server side!

Here is what the client-side code might look like:

Listing 11. 9. The Contacts Table

```
<button hx-post="/integrations/1"> ❶
  Pull Contacts From Integration
</button>

   ...

 <table hx-get="/contacts/table" hx-trigger="contacts-updated from:body"> ❷
   ...
 </table>
```

❶ The response to this request may conditionally trigger the `contacts-updated` event

❷ This table listens for the event and refreshes when it occurs

The table listens for the `contacts-updated` event, and it does so on the `body` element. It listens on the `body` element since the event will bubble up from the button, and this allows us to not couple the button and table together: we can move the button and table around as we like and, via events, the behavior we want will continue to work fine. Additionally, we may want *other* elements or requests to trigger the `contacts-updated` event, so this provides a general mechanism for refreshing the contacts table in our application. Very nice!

Now, we are omitting the server side implementation of this feature in the interest of simplicity, but this gives you and idea of how the `HX-Trigger` response header can be used to coordinate sophisticated interactions in the DOM.

11.3. HTTP Requests & Responses

We have just seen an advanced feature of HTTP responses supported by htmx, the `HX-Trigger` response header, but htmx supports quite a few more headers for both requests and responses. In chapter 5 we discussed the headers present in HTTP Requests. Here some of the more important headers you can use to change htmx behavior with HTTP responses:

Response Header	Description
`HX-Location`	Causes a client-side redirection to a new location
`HX-Push-Url`	Pushes a new URL into the location bar
`HX-Refresh`	Refreshes the current page
`HX-Retarget`	Allows you to specify a new target to swap the response content into on the client side

You can find a reference for all requests and response headers here: https://htmx.org/reference/#headers

11.3.1. HTTP Response Codes

Even more important than response headers, in terms of information conveyed to the client, is the *HTTP Response Code*. We discussed HTTP Response Codes in Chapter 4. By and

large htmx handles various response codes in the manner that you would expect: it swaps content for all 200-level response codes and does nothing for others. There are, however, two "special" 200-level response codes:

- `204 No Content` - When htmx receives this response code, it will *not* swap any content into the DOM (even if the response has a body)
- `286` - When htmx receives this response code to a request that is polling, it will stop the polling

You can override the behavior of htmx with respect to response codes by, you guessed it, responding to an event! The `htmx:beforeSwap` event allows you to change the behavior of htmx with respect to various status codes.

Let's say that, rather than doing nothing when a `404` occurred, you wanted to alert the user that an error had occurred. To do so, you want to invoke a JavaScript method, `showNotFoundError()`. Let's add some code to use the `htmx:beforeSwap` event to make this happen:

Listing 11. 10. Showing a 404 Dialog

```
document.body.addEventListener('htmx:beforeSwap', function(evt) { ❶
    if(evt.detail.xhr.status === 404){ ❷
        showNotFoundError();
    }
});
```

❶ hook into the `htmx:beforeSwap` event

❷ if the response code is a `404`, show the user a dialog

You can also use the `htmx:beforeSwap` event to configure if the response should be swapped into the DOM and what element the response should target. This gives you quite a bit of flexibility in choosing how you want to use HTTP Response codes in your application. Full documentation on the `htmx:beforeSwap` event can be found here: https://htmx.org/events/#htmx:beforeSwap

11.4. Updating Other Content

Above we saw how to use a server-triggered event, via the `HX-Trigger` HTTP response

header, to update a piece of the DOM based on the response to another part of the DOM. This technique addresses the general problem that comes up in Hypermedia Driven Applications: "How do I update other content?" After all, in normal HTTP requests, there is only one "target", the entire screen, and, similarly, in htmx-based requests, there is only one target: either the explicit or implicit target of the element.

If you want to update other content in htmx, you have a few options:

11.4.1. Expanding Your Selection

The first option, and the simplest, is to "expand the target". That is, rather than simply replacing a small part of the screen, expand the target of your htmx-driven request until it is large enough to enclose all the elements that need to updated on a screen. This has the tremendous advantage of being simple and reliable. The downside is that it may not provide the user experience that you want, and it may not play well with a particular server-side template layout. Regardless, I always recommend at least thinking about this approach first.

11.4.2. Out of Band Swaps

A second option, which is a bit more complex, is to take advantage of "Out Of Band" content support in htmx. When htmx receives a response, it will look for top-level content in that response that includes the `hx-swap-oob` attribute on it. That content will be removed from the response, so it will not be swapped into the DOM in the normal manner. Instead, it will be swapped in for the content that it matches, by its id.

Let’s look at an example of this approach. Let’s consider the situation we had above, where a contacts table needs to be updated conditionally, based on if an integration pulls down any new contacts. Previously we solved this by using events and a server-triggered event via the `HX-Trigger` response header.

In this case, instead of using an event, let’s take advantage of the `hx-swap-oob` attribute in the response to the `POST` to `/integrations/1` to "piggy back" the new contacts table content on the response.

Listing 11. 11. The Updated Contacts Table

```
<button hx-post="/integrations/1"> ❶
  Pull Contacts From Integration
</button>

   ...

 <table id="contacts-table"> ❷
   ...
 </table>
```

❶ the button still issues a `POST` to `/integrations/1`

❷ the table no longer listens for an event, but it now has an id

Now let's look at a potential response to the `POST` to `/integrations/1`. This response will include the "regular" content that needs to be swapped into the button, per the usual htmx mechanism. But it will also include a new version, updated version of the contacts table, which will be marked as `hx-swap-oob="true"`. This content will be removed from the response so it is not inserted into the button, but will be instead swapped into the DOM in place of the existing table since it has the same id value.

Listing 11. 12. A Response With Out of Band Content

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...

Pull Contacts From Integration ❶

<table id="contacts-table" hx-swap-oob="true"> ❷
  ...
</table>
```

❶ this content will be placed in the button

❷ this content will be removed from the response and swapped by id

Using this technique, you are able to piggyback content updates of other elements on top of requests by other elements. The `hx-swap-oob` attribute supports other additional features, all of which are documented here: https://htmx.org/attributes/hx-swap-oob/

Depending on how exactly your server side templating technology works, and what level of interactivity your application requires, out of band swapping can be a powerful mechanism for more flexible content updates.

11.4.3. Events

Finally, the most complex mechanism for updating content is the one we saw back in the events section: using server-triggered events to update elements. This approach can be very clean, but also requires a lot deeper conceptual knowledge of HTML and events, and a commitment to the event-driven approach. While we like this style of development, it isn't for everyone and we typically recommend this only if the htmx philosophy of event-driven hypermedia really speaks to you.

If it *does* speak to you, however, we say: go for it! We've created some very complex and flexible user interfaces using this approach, and we are quite fond of it.

11.4.4. Being Pragmatic

All of these approaches to the "Updating Other Content" problem will work, and will often work well. However, there may come a point where it would just be simpler to use a different approach, like the reactive one. As much as we like the hypermedia approach, the reality is that there are some UX patterns that simply cannot be implemented easily using it. The canonical example of this sort of pattern, which we have mentioned before, is something like a live online spreadsheet: it is simply to complex a user interface, with too many inter-dependencies, to be done well via exchanges of hypermedia with a server.

In cases like this, and any time you feel like an htmx-based solution is proving to be more complex than another approach might be, we can gladly recommend that you consider a different technology: use the right tool for the job! You can always use htmx for the parts of your application that aren't as complex and don't need the full complexity of a reactive framework, and save that complexity budget for the parts that do.

We are not hypermedia puritans and encourage you to learn many different web technologies, with an eye to the strengths and weaknesses of each one. This will give you a deep tool chest to reach into when problems present themselves. Our hope is that, with htmx, hypermedia might be a tool you reach for more frequently!

11.5. Debugging

We have been talking a lot about events in this chapter and we are not ashamed to admit: we are big fans of events. They are the underlying technology of almost any interesting user interface, and are particularly useful in the DOM once they have been unlocked for general using in HTML. They let you build nicely decoupled software while often preserving the locality of behavior we like so much.

However, events are not perfect. One area where events can be particularly tricky to deal with is *debugging*: you often want to know why an event *isn't* happening. But where can you set a break point for something that *isn't* happening? The answer, as of right now, is: you can't.

There are two techniques that can help in this regard, one provided by htmx, the other provided by Chrome, the browser by google.

11.5.1. Logging htmx Events

The first technique, provided by htmx itself, is to call the `htmx.logAll()` method. When you do this, htmx will log all the internal events that occur as it goes about its business, loading up content, responding to events and so forth.

This can be overwhelming, but with judicious filtering can help you zero in on a problem. Here are what (a bit of) the logs look like when clicking on the "docs" link on https://htmx.org, with `logAll()` enabled:

Listing 11. 13. htmx Logs

```
htmx:configRequest
<a href="/docs/">
Object { parameters: {}, unfilteredParameters: {}, headers: {…}, target: body, verb:
"get", errors: [], withCredentials: false, timeout: 0, path: "/docs/",
triggeringEvent: a
, … }
htmx.js:439:29
htmx:beforeRequest
<a href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {}, pathInfo:
{…}, elt: a
 }
htmx.js:439:29
htmx:beforeSend
<a class="htmx-request" href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {}, pathInfo:
{…}, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:loadstart
<a class="htmx-request" href="/docs/">
Object { lengthComputable: false, loaded: 0, total: 0, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:progress
<a class="htmx-request" href="/docs/">
Object { lengthComputable: true, loaded: 4096, total: 19915, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:progress
<a class="htmx-request" href="/docs/">
Object { lengthComputable: true, loaded: 19915, total: 19915, elt: a.htmx-request
 }
htmx.js:439:29
htmx:beforeOnLoad
<a class="htmx-request" href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {}, pathInfo:
{…}, elt: a.htmx-request
 }
htmx.js:439:29
htmx:beforeSwap
<body hx-ext="class-tools, preload">
```

Yikes! Not exactly easy on the eyes, is it? But, if you take a deep breath and squint, you can see that it isn't *that* bad: a series of htmx events, some of which we have seen before (there's `htmx:configRequest`!), get logged to the console, along with the element they are triggered on. After a bit of reading and filtering, you will be able to make sense of the event stream, and it can help you debug htmx-related issues.

11.5.2. Monitoring Events in Chrome

The preceding technique is useful if the problem is occuring somewhere *within* htmx, but what if htmx is never getting triggered at all? This comes up some times, like when, for example, you have accidentally typed an event name incorrectly somewhere.

In cases like this you will need recourse to a tool available in the browser itself. Fortunately, the Chrome browser by Google provides a very useful function, `monitorEvents()`, that allows you to monitor *all* events that are triggered on an element. This feature is available *only* in the console, so you can't use it in code on your page. But, if you are working with htmx in Chrome, and are curious why an event isn't triggering on an element, you can open the developers console and type the following:

Listing 11. 14. htmx Logs

```
monitorEvents(document.getElementById("some-element"));
```

This will then print *all* the events that are triggered on the element with the id `some-element` to the console. This can be very useful for understanding exactly which events you want to respond to with htmx, or troubleshooting why an expected event isn't occurring.

Using these two techniques will help you as you (infrequently, we hope!) troubleshoot event-related issues when developing with htmx.

11.6. Security Considerations

In general, htmx and hypermedia tends to be more secure than JavaScript heavy approaches to building web applications. This is because, by moving much of the processing to the back end, the hypermedia approach tends not to expose as much surface area of your system to end users for manipulation and shenanigans.

However, even with hypermedia, there are still situations that require care when doing

development. Of particular concern are situations where user-generated content is shown to other users: a clever user might try to insert htmx code that tricks the other users into clicking on content that triggers actions they don't want to take.

In general, all user-generated content should be escaped on the server side, and most server side rendering frameworks provide functionality for handling this situation. But there is always a risk that something slips through the cracks.

In order to help you sleep better at night, htmx provides the `hx-disable` attribute. When this attribute is placed on an element, all htmx attributes within that element will be ignored.

11.6.1. Content Security Policies & htmx

A Content Security Policy (CSP) is a browser technology that allows you to detect and prevent certain types of content injection-based attacks. A full discussion of CSPs is beyond the scope of this book, but we refer you to the Mozilla Developer Network article on them for more information: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

A common feature to disable using a CSP is the `eval()` feature of JavaScript, which allows you to evaluate arbitrary javascript code from a string. This has proven to be a security issue and many teams have decided that it is not worth the risk to keep it enabled in their web applications.

htmx does not make heavy use of `eval()` and, thus, a CSP with this restriction in place will be fine. The one feature that does rely on `eval()` is event filters, discussed above. If you decide to disable `eval()` for your web application, you will not be able to use the event filtering syntax.

11.7. Configuring

There are a large number of configuration options available for htmx. Some examples of things you can configure are:

- The default swap style
- The default swap delay
- The default timeout of AJAX requests

A full list of configuration options can be found in the config section of the main htmx

documentation: https://htmx.org/docs/#config

htmx is typically configured via a `meta` tag, found in the header of a page. The name of the meta tag should be `htmx-config`, and the content attribute should contain the configuration overrides, formatted as JSON. Here is an example:

Listing 11. 15. An htmx configuration via a `meta` tag

```
<meta name="htmx-config" content='{"defaultSwapStyle":"outerHTML"}'>
```

In this case, we are overriding the default swap style from the usual `innerHTML` to `outerHTML`. This might be useful if you find yourself using `outerHTML` more frequently than `innerHTML` and want to avoid having to explicitly set that swap value throughout your application.

11.8. Summary

- In this chapter we looked at some details and tricks of htmx development
- We looked in detail at the options available for `hx-swap` and `hx-trigger`, including filters and scrolling
- We took a look at the events that htmx triggers and responds to
- We explored HTTP response headers and HTTP response codes in htmx, and how to modify how htmx handles them
- We looked at various techniques for updating content beyond the target of a request, including out of band swaps
- We saw how htmx applications can be debugged, secured and configured