Contents

Htmx in a Nutshell

Htmx is a library that allows you to access modern browser features directly from HTML, rather than using javascript.

To understand htmx, first lets take a look at an anchor tag:

```
<a href="/blog">Blog</a>
```

This anchor tag tells a browser:

> "When a user clicks on this link, issue an HTTP GET request to '/blog' and load the response content into the browser window".

With that in mind, consider the following bit of HTML:

```
<button hx-post="/clicked"
    hx-trigger="click"
    hx-target="#parent-div"
    hx-swap="outerHTML"
>
    Click Me!
</button>
```

This tells htmx:

> "When a user clicks on this button, issue an HTTP POST request to '/clicked' and use the content from the response to replace the element with the id `parent-div` in the DOM"

Htmx extends and generalizes the core idea of HTML as a hypertext, opening up many more possibilities directly within the language:

- Now any element, not just anchors and forms, can issue an HTTP request
- Now any event, not just clicks or form submissions, can trigger requests
- Now any HTTP verb, not just `GET` and `POST`, can be used
- Now any element, not just the entire window, can be the target for update by the request

Note that when you are using htmx, on the server side you typically respond with *HTML*, not *JSON*. This keeps you firmly within the original web programming model, using Hypertext As The Engine Of Application State without even needing to really understand that concept.

It's worth mentioning that, if you prefer, you can use the `data-` prefix when using htmx:

```
<a data-hx-post="/click">Click Me!</a>
```

Installing

Htmx is a dependency-free, browser-oriented javascript library. This means that using it is as simple as adding a `<script>` tag to your document head. No need for complicated build steps or systems.

If you are migrating to htmx from intercooler.js, please see the migration guide.

Via A CDN (e.g. unpkg.com)

The fastest way to get going with htmx is to load it via a CDN. You can simply add this to your head tag and get going:

```
<script src="https://unpkg.com/htmx.org@1.8.0" integrity="sha384-cZ
```

While the CDN approach is extremely simple, you may want to consider not using CDNs in production.

Download a copy

The next easiest way to install htmx is to simply copy it into your project.

Download `htmx.min.js` from unpkg.com and add it to the appropriate directory in your project and include it where necessary with a `<script>` tag:

```
<script src="/path/to/htmx.min.js"></script>
```

You can also add extensions this way, by downloading them from the `ext/` directory.

npm

For npm-style build systems, you can install htmx via npm:

```
npm install htmx.org
```

After installing, you'll need to use appropriate tooling to use `node_modules/htmx.org/dist/htmx.js` (or `.min.js`). For example, you might bundle htmx with some extensions and project-specific code.

Webpack

If you are using webpack to manage your javascript:

- Install `htmx` via your favourite package manager (like npm or yarn)
- Add the import to your `index.js`

```
import 'htmx.org';
```

If you want to use the global `htmx` variable (recommended), you need to inject it to the window scope:

- Create a custom JS file
- Import this file to your `index.js` (below the import from step 2)

```
import 'path/to/my_custom.js';
```

- Then add this code to the file:

```
window.htmx = require('htmx.org');
```

- Finally, rebuild your bundle

AJAX

The core of htmx is a set of attributes that allow you to issue AJAX requests directly from HTML:

Attribute	Description
hx-get	Issues a `GET` request to the given URL
hx-post	Issues a `POST` request to the given URL
hx-put	Issues a `PUT` request to the given URL
hx-patch	Issues a `PATCH` request to the given URL
hx-delete	Issues a `DELETE` request to the given URL

Each of these attributes takes a URL to issue an AJAX request to. The element will issue a request of the specified type to the given URL when the element is triggered:

```
<div hx-put="/messages">
    Put To Messages
</div>
```

This tells the browser:

> When a user clicks on this div, issue a PUT request to the URL /messages and load the response into the div

Triggering Requests

By default, AJAX requests are triggered by the "natural" event of an element:

- `input`, `textarea` & `select` are triggered on the `change` event
- `form` is triggered on the `submit` event
- everything else is triggered by the `click` event

If you want different behavior you can use the hx-trigger attribute to specify which event will cause the request.

Here is a `div` that posts to `/mouse_entered` when a mouse enters it:

```
<div hx-post="/mouse_entered" hx-trigger="mouseenter">
    [Here Mouse, Mouse!]
</div>
```

Trigger Modifiers

A trigger can also have a few additional modifiers that change its behavior. For example, if you want a request to only happen once, you can use the `once` modifier for the trigger:

```
<div hx-post="/mouse_entered" hx-trigger="mouseenter once">
    [Here Mouse, Mouse!]
```

```
</div>
```

Other modifiers you can use for triggers are:

- `changed` - only issue a request if the value of the element has changed
- `delay:<time interval>` - wait the given amount of time (e.g. `1s`) before issuing the request. If the event triggers again, the countdown is reset.
- `throttle:<time interval>` - wait the given amount of time (e.g. `1s`) before issuing the request. Unlike `delay` if a new event occurs before the time limit is hit the event will be discarded, so the request will trigger at the end of the time period.
- `from:<CSS Selector>` - listen for the event on a different element. This can be used for things like keyboard shortcuts.

You can use these attributes to implement many common UX patterns, such as Active Search:

```
<input type="text" name="q"
    hx-get="/trigger_delay"
    hx-trigger="keyup changed delay:500ms"
    hx-target="#search-results"
    placeholder="Search..."
>
<div id="search-results"></div>
```

This input will issue a request 500 milliseconds after a key up event if the input has been changed and inserts the results into the `div` with the id `search-results` .

Multiple triggers can be specified in the hx-trigger attribute, separated by commas.

Trigger Filters

You may also apply trigger filters by using square brackets after the event name, enclosing a javascript expression that will be evaluated. If the expression evaluates to `true` the event will trigger, otherwise it will not.

Here is an example that triggers only on a Control-Click of the element

```
<div hx-get="/clicked" hx-trigger="click[ctrlKey]">
    Control Click Me
</div>
```

Properties like `ctrlKey` will be resolved against the triggering event first, then the global scope.

Special Events

htmx provides a few special events for use in hx-trigger:

- `load` - fires once when the element is first loaded
- `revealed` - fires once when an element first scrolls into the viewport
- `intersect` - fires once when an element first intersects the viewport. This supports two additional options:
  - `root:<selector>` - a CSS selector of the root element for intersection
  - `threshold:<float>` - a floating point number between 0.0 and 1.0, indicating what amount of intersection to fire the event on

You can also use custom events to trigger requests if you have an advanced use case.

Polling

If you want an element to poll the given URL rather than wait for an event, you can use the `every` syntax with the `hx-trigger` attribute:

```
<div hx-get="/news" hx-trigger="every 2s"></div>
```

This tells htmx

> Every 2 seconds, issue a GET to /news and load the response into the div

If you want to stop polling from a server response you can respond with the HTTP response code `286` and the element will cancel the polling.

Load Polling

Another technique that can be used to achieve polling in htmx is "load polling", where an element specifies a `load` trigger along with a delay, and replaces itself with the response:

```
<div hx-get="/messages"
     hx-trigger="load delay:1s"
     hx-swap="outerHTML"
>
</div>
```

If the `/messages` end point keeps returning a div set up this way, it will keep "polling" back to the URL every second.

Load polling can be useful in situations where a poll has an end point at which point the polling terminates, such as when you are showing the user a progress bar.

Request Indicators

When an AJAX request is issued it is often good to let the user know that something is happening since the browser will not give them any feedback. You can accomplish this in htmx by using `htmx-indicator` class.

The `htmx-indicator` class is defined so that the opacity of any element with this class is 0 by default, making it invisible but present in the DOM.

When htmx issues a request, it will put a `htmx-request` class onto an element (either the requesting element or another element, if specified). The `htmx-request` class will cause a child element with the `htmx-indicator` class on it to transition to an opacity of 1, showing the indicator.

```
<button hx-get="/click">
    Click Me!
    <img class="htmx-indicator" src="/spinner.gif">
</button>
```

Here we have a button. When it is clicked the `htmx-request` class will be added to it, which will reveal the spinner gif element. (I like SVG spinners these days.)

While the `htmx-indicator` class uses opacity to hide and show the progress indicator, if you would prefer another mechanism you can create your own CSS transition like so:

```
.htmx-indicator{
    display:none;
}
.htmx-request .my-indicator{
    display:inline;
}
.htmx-request.my-indicator{
    display:inline;
}
```

If you want the `htmx-request` class added to a different element, you can use the hx-indicator attribute with a CSS selector to do so:

```
<div>
    <button hx-get="/click" hx-indicator="#indicator">
        Click Me!
```

```
    </button>
    <img id="indicator" class="htmx-indicator" src="/spinner.gif"/>
</div>
```

Here we call out the indicator explicitly by id. Note that we could have placed the class on the parent `div` as well and had the same effect.

Targets

If you want the response to be loaded into a different element other than the one that made the request, you can use the hx-target attribute, which takes a CSS selector. Looking back at our Live Search example:

```
<input type="text" name="q"
    hx-get="/trigger_delay"
    hx-trigger="keyup delay:500ms changed"
    hx-target="#search-results"
    placeholder="Search..."
>
<div id="search-results"></div>
```

You can see that the results from the search are going to be loaded into `div#search-results`, rather than into the input tag.

Swapping

htmx offers a few different ways to swap the HTML returned into the DOM. By default, the content replaces the `innerHTML` of the target element. You can modify this by using the hx-swap attribute with any of the following values:

Name	Description
`innerHTML`	the default, puts the content inside the target element

Name	Description
`outerHTML`	replaces the entire target element with the returned content
`afterbegin`	prepends the content before the first child inside the target
`beforebegin`	prepends the content before the target in the targets parent element
`beforeend`	appends the content after the last child inside the target
`afterend`	appends the content after the target in the targets parent element
`none`	does not append content from response (Out of Band Swaps and Response Headers will still be processed)

Synchronization

Often you want to coordinate the requests between two elements. For example, you may want a request from one element to supersede the request of another element, or to wait until the other elements request has finished.

htmx offers a `hx-sync` attribute to help you accomplish this.

Consider a race condition between a form submission and an individual input's validation request in this HTML:

```
<form hx-post="/store">
    <input id="title" name="title" type="text"
        hx-post="/validate"
        hx-trigger="change"
    >
    <button type="submit">Submit</button>
</form>
```

Without using `hx-sync`, filling out the input and immediately submitting the form triggers two parallel requests to `/validate` and `/store`.

Using `hx-sync="closest form:abort"` on the input will watch for requests on the form and abort the input's request if a form request is present or starts while the input request is in flight:

```
<form hx-post="/store">
    <input id="title" name="title" type="text"
        hx-post="/validate"
        hx-trigger="change"
        hx-sync="closest form:abort"
    >
    <button type="submit">Submit</button>
</form>
```

This resolves the synchronization between the two elements in a declarative way.

htmx also supports a programmatic way to cancel requests: you can send the `htmx:abort` event to an element to cancel any in-flight requests:

```
<button id="request-button" hx-post="/example">
    Issue Request
</button>
<button onclick="htmx.trigger('#request-button', 'htmx:abort')">
    Cancel Request
</button>
```

More examples and details can be found on the `hx-sync` attribute page.

CSS Transitions

htmx makes it easy to use CSS Transitions without javascript. Consider this HTML content:

```
<div id="div1">Original Content</div>
```

Imagine this content is replaced by htmx via an ajax request with this new content:

```
<div id="div1" class="red">New Content</div>
```

Note two things:

- The div has the *same* id in the original an in the new content
- The `red` class has been added to the new content

Given this situation, we can write a CSS transition from the old state to the new state:

```
.red {
    color: red;
    transition: all ease-in 1s ;
}
```

When htmx swaps in this new content, it will do so in such a way that the CSS transition will apply to the new content, giving you a nice, smooth transition to the new state.

So, in summary, all you need to do to use CSS transitions for an element is keep its `id` stable across requests!

You can see the Animation Examples for more details and live demonstrations.

Details

To understand how CSS transitions actually work in htmx, you must understand the underlying swap & settle model that htmx uses.

When new content is received from a server, before the content is swapped in, the existing content of the page is examined for elements that match by the `id` attribute. If a match is found for an element in the new content, the attributes of the old content are copied onto the new element before the swap occurs. The new content is then swapped in, but with the *old* attribute values. Finally, the new attribute values are swapped in, after a "settle" delay (20ms by default). A little crazy, but this is what allowes CSS transitions to work without any javascript by the developer.

Out of Band Swaps

If you want to swap content from a response directly into the DOM by using the `id` attribute you can use the hx-swap-oob attribute in the *response* html:

```
<div id="message" hx-swap-oob="true">Swap me directly!</div>
Additional Content
```

In this response, `div#message` would be swapped directly into the matching DOM element, while the additional content would be swapped into the target in the normal manner.

You can use this technique to "piggy-back" updates on other requests.

Note that out of band elements must be in the top level of the response, and not children of the top level elements.

Selecting Content To Swap

If you want to select a subset of the response HTML to swap into the target, you can use the hx-select attribute, which takes a CSS selector and selects the matching elements from the response.

You can also pick out pieces of content for an out-of-band swap by using the hx-select-oob attribute, which takes a list of element IDs to pick out and swap.

Preserving Content During A Swap

If there is content that you wish to be preserved across swaps (e.g. a video player that you wish to remain playing even if a swap occurs) you can use the hx-preserve attribute on the elements you wish to be preserved.

Parameters

By default, an element that causes a request will include its value if it has one. If the element is a form it will include the values of all inputs within it.

Additionally, if the element causes a non- `GET` request, the values of all the inputs of the nearest enclosing form will be included.

If you wish to include the values of other elements, you can use the hx-include attribute with a CSS selector of all the elements whose values you want to include in the request.

If you wish to filter out some parameters you can use the hx-params attribute.

Finally, if you want to programatically modify the parameters, you can use the htmx:configRequest event.

File Upload

If you wish to upload files via an htmx request, you can set the hx-encoding attribute to `multipart/form-data`. This will use a `FormData` object to submit the request, which will properly include the file in the request.

Note that depending on your server-side technology, you may have to handle requests with this type of body content very differently.

Note that htmx fires a `htmx:xhr:progress` event periodically based on the standard `progress` event during upload, which you can hook into to show the progress of the upload.

Extra Values

You can include extra values in a request using the hx-vals (name-expression pairs in JSON format) and hx-vars attributes (comma-separated name-expression pairs that are dynamically computed).

Confirming Requests

Often you will want to confirm an action before issuing a request. htmx supports the `hx-confirm` attribute, which allows you to confirm an action using a simple javascript dialog:

```
<button hx-delete="/account" hx-confirm="Are you sure you wish to de
    Delete My Account
</button>
```

Using events you can implement more sophisticated confirmation dialogs. The confirm example shows how to use sweetalert2 library for confirmation of htmx actions.

Attribute Inheritance

Most attributes in htmx are inherited: they apply to the element they are on as well as any children elements. This allows you to "hoist" attributes up the DOM to avoid code duplication. Consider the following htmx:

```
<button hx-delete="/account" hx-confirm="Are you sure?">
    Delete My Account
</button>
<button hx-put="/account" hx-confirm="Are you sure?">
    Update My Account
</button>
```

Here we have a duplicate `hx-confirm` attribute. We can hoist this attribute to a parent element:

```
<div hx-confirm="Are you sure?">
    <button hx-delete="/account">
        Delete My Account
    </button>
    <button hx-put="/account">
        Update My Account
    </button>
</div>
```

This `hx-confirm` attribute will now apply to all htmx-powered elements within it.

Sometimes you wish to undo this inheritance. Consider if we had a cancel button to this group, but didn't want it to be confirmed. We could add an `unset` directive on it like so:

```
<div hx-confirm="Are you sure?">
    <button hx-delete="/account">
        Delete My Account
    </button>
    <button hx-put="/account">
        Update My Account
    </button>
    <button hx-confirm="unset" hx-get="/">
        Cancel
    </button>
</div>
```

The top two buttons would then show a confirm dialog, but the bottom cancel button would not.

Automatic inheritance can be further d using `hx-disinherit` attribute.

Boosting

Htmx supports "boosting" regular HTML anchors and forms with the hx-boost attribute. This attribute will convert all anchor tags and forms into AJAX requests that, by default, target the body of the page.

Here is an example:

```
<div hx-boost="true">
    <a href="/blog">Blog</a>
</div>
```

The anchor tag in this div will issue an AJAX `GET` request to `/blog` and swap the response into the `body` tag.

Progressive Enhancement

A feature of `hx-boost` is that it degrades gracefully if javascript is not enabled: the links and forms continue to work, they simply don't use ajax requests. This is known as Progressive Enhancement, and it allows a wider audience to use your sites functionality.

Other htmx patterns can be adapted to achieve progressive enhancement as well, but they will require more thought.

Consider the active search example. As it is written, it will not degrade gracefully: someone who does not have javascript enabled will not be able to use this feature. This is done for simplicities sake, to keep the example as brief as possible.

However, you could wrap the htmx-enhanced input in a form element:

```
<form action="/search" method="POST">
```

```
    <input class="form-control" type="search"
        name="search" placeholder="Begin typing to search users..."
        hx-post="/search"
        hx-trigger="keyup changed delay:500ms, search"
        hx-target="#search-results"
        hx-indicator=".htmx-indicator"
    >
</form>
```

With this in place, javascript-enabled clients would still get the nice active-search UX, but non-javascript enabled clients would be able to hit the enter key and still search. Even better, you could add a "Search" button as well. You would then need to update the form with an `hx-post` that mirrored the `action` attribute, or perhaps use `hx-boost` on it.

You would need to check on the server side for the `HX-Request` header to differentiate between an htmx-driven and a regular request, to determine exactly what to render to the client.

Other patterns can be adapted similarly to achieve the progressive enhancement needs of your application.

As you can see, this requires more thought and more work. It also rules some functionality entirely out of bounds. These tradeoffs must be made by you, the developer, with respect to your projects goals and audience.

Accessibility is a concept closely related to progressive enhancement. Using progressive enhancement techniques such as `hx-boost` will make your htmx application more accessible to a wide array of users.

htmx-based applications are very similar to normal, non-AJAX driven web applications because htmx is HTML-oriented.

As such, the normal HTML accessibility recommendations apply. For example:

- Use semantic HTML as much as possible (i.e. the right tags for the right things)
- Ensure focus state is clearly visible
- Associate text labels with all form fields
- Maximize the readability of your application with appropriate fonts, contrast, etc.

Web Sockets & SSE

htmx has experimental support for declarative use of both WebSockets and Server Sent Events.

> **Note:** In htmx 2.0, these features will be migrated to extensions. These new extensions are already available in htmx 1.7+ and, if you are writing new code, you are encouraged to use the extensions instead. All new feature work for both SSE and web sockets will be done in the extensions.
>
> Please visit the SSE extension and WebSocket extension pages to learn more about the new extensions.

WebSockets

If you wish to establish a `WebSocket` connection in htmx, you use the hx-ws attribute:

```
<div hx-ws="connect:wss:/chatroom">
    <div id="chat_room">
        ...
    </div>
    <form hx-ws="send:submit">
        <input name="chat_message">
```

```
    </form>
</div>
```

The `connect` declaration established the connection, and the `send` declaration tells the form to submit values to the socket on `submit`.

More details can be found on the hx-ws attribute page

Server Sent Events

Server Sent Events are a way for servers to send events to browsers. It provides a higher-level mechanism for communication between the server and the browser than websockets.

If you want an element to respond to a Server Sent Event via htmx, you need to do two things:

1. Define an SSE source. To do this, add a hx-sse attribute on a parent element with a `connect <url>` declaration that specifies the URL from which Server Sent Events will be received.
2. Define elements that are descendents of this element that are triggered by server sent events using the `hx-trigger="sse:<event_name>"` syntax

Here is an example:

```
<body hx-sse="connect:/news_updates">
    <div hx-trigger="sse:new_news" hx-get="/news"></div>
</body>
```

Depending on your implementation, this may be more efficient than the polling example above since the server would notify the div if there was new news to get, rather than the steady requests that a poll causes.

History Support

Htmx provides a simple mechanism for interacting with the browser history API:

If you want a given element to push its request URL into the browser navigation bar and add the current state of the page to the browser's history, include the hx-push-url attribute:

```
<a hx-get="/blog" hx-push-url="true">Blog</a>
```

When a user clicks on this link, htmx will snapshot the current DOM and store it before it makes a request to /blog. It then does the swap and pushes a new location onto the history stack.

When a user hits the back button, htmx will retrieve the old content from storage and swap it back into the target, simulating "going back" to the previous state. If the location is not found in the cache, htmx will make an ajax request to the given URL, with the header `HX-History-Restore-Request` set to true, and expects back the HTML needed for the entire page. Alternatively, if the `htmx.config.refreshOnHistoryMiss` config variable is set to true, it will issue a hard browser refresh.

NOTE: If you push a URL into the history, you **must** be able to navigate to that URL and get a full page back! A user could copy and paste the URL into an email, or new tab. Additionally, htmx will need the entire page when restoring history if the page is not in the history cache.

Specifying History Snapshot Element

By default, htmx will use the `body` to take and restore the history snapshot from. This is usually the right thing, but if you want to use a narrower element for snapshotting you can use the hx-history-elt attribute to specify a different one.

Careful: this element will need to be on all pages or restoring from history won't work reliably.

Requests & Responses

Htmx expects responses to the AJAX requests it makes to be HTML, typically HTML fragments (although a full HTML document, matched with a hx-select tag can be useful too). Htmx will then swap the returned HTML into the document at the target specified and with the swap strategy specified.

Sometimes you might want to do nothing in the swap, but still perhaps trigger a client side event (see below). For this situation you can return a `204 - No Content` response code, and htmx will ignore the content of the response.

In the event of an error response from the server (e.g. a 404 or a 501), htmx will trigger the `htmx:responseError` event, which you can handle.

In the event of a connection error, the `htmx:sendError` event will be triggered.

CORS

When using htmx in a cross origin context, remember to configure your web server to set Access-Control headers in order for htmx headers to be visible on the client side.

- Access-Control-Allow-Headers (for request headers)
- Access-Control-Expose-Headers (for response headers)

See all the request and response headers that htmx implements.

Request Headers

htmx includes a number of useful headers in requests:

Header	Description
`HX-Request`	will be set to "true"
`HX-Trigger`	will be set to the id of the element that triggered the request

Header	Description
`HX-Trigger-Name`	will be set to the name of the element that triggered the request
`HX-Target`	will be set to the id of the target element
`HX-Prompt`	will be set to the value entered by the user when prompted via hx-prompt

Response Headers

htmx supports some htmx-specific response headers:

- `HX-Push` - pushes a new URL into the browser's address bar
- `HX-Redirect` - triggers a client-side redirect to a new location
- `HX-Location` - triggers a client-side redirect to a new location that acts as a swap
- `HX-Refresh` - if set to "true" the client side will do a full refresh of the page
- `HX-Trigger` - triggers client side events
- `HX-Trigger-After-Swap` - triggers client side events after the swap step
- `HX-Trigger-After-Settle` - triggers client side events after the settle step

For more on the `HX-Trigger` headers, see `HX-Trigger` Response Headers.

Submitting a form via htmx has the benefit, that the Post/Redirect/Get Pattern is not needed any more. After successful processing a POST request on the server, you don't need to return a HTTP 302 (Redirect). You can directly return the new HTML fragment.

Request Order of Operations

The order of operations in a htmx request are:

- The element is triggered and begins a request
  - Values are gathered for the request

    - The `htmx-request` class is applied to the appropriate elements
    - The request is then issued asynchronously via AJAX
        - Upon getting a response the target element is marked with the `htmx-swapping` class
        - An optional swap delay is applied (see the hx-swap attribute)
        - The actual content swap is done
            - the `htmx-swapping` class is removed from the target
            - the `htmx-added` class is added to each new piece of content
            - the `htmx-settling` class is applied to the target
            - A settle delay is done (default: 20ms)
            - The DOM is settled
            - the `htmx-settling` class is removed from the target
            - the `htmx-added` class is removed from each new piece of content

You can use the `htmx-swapping` and `htmx-settling` classes to create CSS transitions between pages.

Validation

Htmx integrates with the HTML5 Validation API and will not issue a request if a validatable input is invalid. This is true for both AJAX requests as well as WebSocket sends.

Htmx fires events around validation that can be used to hook in custom validation and error handling:

- `htmx:validation:validate` - called before an elements `checkValidity()` method is called. May be used to add in custom validation logic
- `htmx:validation:failed` - called when `checkValidity()` returns false, indicating an invalid input
- `htmx:validation:halted` - called when a request is not issued due to validation errors. Specific errors may be found in the `event.detail.errors` object

Validation Example

Here is an example of an input that uses the `htmx:validation:validate` event to require that an input have the value `foo`, using hyperscript:

```
<form hx-post="/test">
    <input _="on htmx:validation:validate
                if my.value != 'foo'
                    call me.setCustomValidity('Please enter the valu
                else
                    call me.setCustomValidity('')"
        name="example"
    >
</form>
```

Note that all client side validations must be re-done on the server side, as they can always be bypassed.

Animations

Htmx allows you to use CSS transitions in many situations using only HTML and CSS.

Please see the Animation Guide for more details on the options available.

Extensions

Htmx has an extension mechanism that allows you to customize the libraries' behavior. Extensions are defined in javascript and then used via the `hx-ext` attribute:

```
<div hx-ext="debug">
```

```
    <button hx-post="/example">This button used the debug extension
    <button hx-post="/example" hx-ext="ignore:debug">This button do
</div>
```

If you are interested in adding your own extension to htmx, please see the extension docs

Included Extensions

Htmx includes some extensions that are tested against the htmx code base. Here are a few:

Extension	Description
`json-enc`	use JSON encoding in the body of requests, rather than the default `x-www-form-urlencoded`
`morphdom-swap`	an extension for using the morphdom library as the swapping mechanism in htmx.
`alpine-morph`	an extension for using the Alpine.js morph plugin as the swapping mechanism in htmx.
`client-side-templates`	support for client side template processing of JSON responses
`path-deps`	an extension for expressing path-based dependencies similar to intercoolerjs
`class-tools`	an extension for manipulating timed addition and removal of classes on HTML elements

See the extensions page for a complete list.

Events & Logging

Htmx has an extensive events mechanism, which doubles as the logging system.

If you want to register for a given htmx event you can use

```
document.body.addEventListener('htmx:load', function(evt) {
    myJavascriptLib.init(evt.detail.elt);
});
```

or, if you would prefer, you can use the following htmx helper:

```
htmx.on("htmx:load", function(evt) {
    myJavascriptLib.init(evt.detail.elt);
});
```

The `htmx:load` event is fired every time an element is loaded into the DOM by htmx, and is effectively the equivalent to the normal `load` event.

Some common uses for htmx events are:

Initialize A 3rd Party Library With Events

Using the `htmx:load` event to initialize content is so common that htmx provides a helper function:

```
htmx.onLoad(function(target) {
    myJavascriptLib.init(target);
});
```

This does the same thing as the first example, but is a little cleaner.

Configure a Request With Events

You can handle the `htmx:configRequest` event in order to modify an AJAX request before it is issued:

```
document.body.addEventListener('htmx:configRequest', function(evt) {
    evt.detail.parameters['auth_token'] = getAuthToken(); // add a r
    evt.detail.headers['Authentication-Token'] = getAuthToken(); //
});
```

Here we add a parameter and header to the request before it is sent.

Modifying Swapping Behavior With Events

You can handle the `htmx:beforeSwap` event in order to modify the swap behavior of htmx:

```
document.body.addEventListener('htmx:beforeSwap', function(evt) {
    if(evt.detail.xhr.status === 404){
        // alert the user when a 404 occurs (maybe use a nicer mecha
        alert("Error: Could Not Find Resource");
    } else if(evt.detail.xhr.status === 422){
        // allow 422 responses to swap as we are using this as a sig
        // a form was submitted with bad data and want to rerender w
        // errors
        //
        // set isError to false to avoid error logging in console
        evt.detail.shouldSwap = true;
        evt.detail.isError = false;
    } else if(evt.detail.xhr.status === 418){
        // if the response code 418 (I'm a teapot) is returned, reta
        // content of the response to the element with the id `teapo
        evt.detail.shouldSwap = true;
        evt.detail.target = htmx.find("#teapot");
    }
});
```

Here we handle a few 400-level error response codes that would normally not do a swap in htmx.

Event Naming

Note that all events are fired with two different names

- Camel Case
- Kebab Case

So, for example, you can listen for `htmx:afterSwap` or for `htmx:after-swap`. This facilitates interoperability with other libraries. Alpine.js, for example, requires kebab case.

Logging

If you set a logger at `htmx.logger`, every event will be logged. This can be very useful for troubleshooting:

```
htmx.logger = function(elt, event, data) {
    if(console) {
        console.log(event, elt, data);
    }
}
```

Debugging

Declarative and event driven programming with htmx (or any other declartive language) can be a wonderful and highly productive activity, but one disadvantage when compared with imperative approaches is that it can be trickier to debug.

Figuring out why something *isn't* happening, for example, can be difficult if you don't know the tricks.

Well, here are the tricks:

The first debugging tool you can use is the `htmx.logAll()` method. This will log every event that htmx triggers and will allow you to see exactly what the library is doing.

```
htmx.logAll();
```

Of course, that won't tell you why htmx *isn't* doing something. You might also not know *what* events a DOM element is firing to use as a trigger. To address this, you can use the `monitorEvents()` method available in the browser console:

```
monitorEvents(htmx.find("#theElement"));
```

This will spit out all events that are occuring on the element with the id `theElement` to the console, and allow you to see exactly what is going on with it.

Note that this *only* works from the console, you cannot embed it in a script tag on your page.

Finally, push come shove, you might want to just debug `htmx.js` by loading up the unminimized version. It's about 2500 lines of javascript, so not an insurmountable amount of code. You would most likely want to set a break point in the `issueAjaxRequest()` and `handleAjaxResponse()` methods to see what's going on.

And always feel free to jump on the Discord if you need help.

Creating Demos

Sometimes, in order to demonstrate a bug or clarify a usage, it is nice to be able to use a javascript snippet site like jsfiddle. To facilitate easy demo creation, htmx hosts a demo script site that will install:

- htmx
- hyperscript
- a request mocking library

Simply add the following script tag to your demo/fiddle/whatever:

```
<script src="https://demo.htmx.org"></script>
```

This helper allows you to add mock responses by adding `template` tags with a `url` attribute to indicate which URL. The response for that url will be the innerHTML of the template, making it easy to construct mock responses. You can add a delay to the response with a `delay` attribute, which should be an integer indicating the number of milliseconds to delay

You may embed simple expressions in the template with the `${}` syntax.

Note that this should only be used for demos and is in no way guaranteed to work for long periods of time as it will always be grabbing the latest versions htmx and hyperscript!

Demo Example

Here is an example of the code in action:

```
<!-- load demo environment -->
<script src="https://demo.htmx.org"></script>

<!-- post to /foo -->
<button hx-post="/foo" hx-target="#result">
    Count Up
</button>
<output id="result"></output>
```

```
<!-- respond to /foo with some dynamic content in a template tag -->
<script>
    globalInt = 0;
</script>
<template url="/foo" delay="500"> <!-- note the url and delay attri
    ${globalInt++}
</template>
```

hyperscript

Hyperscript is an experimental front end scripting language designed to be expressive and easily embeddable directly in HTML for handling custom events, etc. The language is inspired by HyperTalk, javascript, gosu and others.

You can explore the language more fully on its main website:

http://hyperscript.org

Hyperscript is *not* required when using htmx, anything you can do in hyperscript can be done in vanilla JS or with another javascript library like jQuery, but the two technologies were designed with one another in mind and play well together.

Installing Hyperscript

To use hyperscript in combination with htmx, you need to install the hyperscript library either via a CDN or locally. See the hyperscript website for the latest version of the library.

When hyperscript is included, it will automatically integrate with htmx and begin processing all hyperscripts embedded in your HTML.

Events & Hyperscript

Hyperscript was designed to help address features and functionality from intercooler.js that are not implemented in htmx directly, in a more flexible and open manner. One of its prime features is the ability to respond to arbitrary events on a DOM element, using the `on` syntax:

```
<div _="on htmx:afterSettle log 'Settled!'">
    ...
</div>
```

This will log `Settled!` to the console when the `htmx:afterSettle` event is triggered.

intercooler.js features & hyperscript implementations

Below are some examples of intercooler features and the hyperscript equivalent.

`ic-remove-after`

Intercooler provided the `ic-remove-after` attribute for removing an element after a given amount of time.

In hyperscript you can implement this, as well as fade effect, like so:

```
<div _="on load wait 5s then transition opacity to 0 then remove me
    Here is a temporary message!
</div>
```

`ic-post-errors-to`

Intercooler provided the `ic-post-errors-to` attribute for posting errors that occured during requests and responses.

In hyperscript similar functionality is implemented like so:

```
<body _="on htmx:error(errorInfo) fetch /errors {method:'POST', body
    ...
</body>
```

ic-switch-class

Intercooler provided the `ic-switch-class` attribute, which let you switch a class between siblings.

In hyperscript you can implement similar functionality like so:

```
<div hx-target="#content" _="on htmx:beforeOnLoad take .active from
    <a class="tabs active" hx-get="/tab1" >Tab 1</a>
    <a class="tabs" hx-get="/tab2">Tab 2</a>
    <a class="tabs" hx-get="/tab3">Tab 3</a>
</div>
<div id="content">Tab 1 Content</div>
```

3rd Party Javascript

Htmx integrates fairly well with third party libraries. If the library fires events on the DOM, you can use those events to trigger requests from htmx.

A good example of this is the SortableJS demo:

```
<form class="sortable" hx-post="/items" hx-trigger="end">
    <div class="htmx-indicator">Updating...</div>
    <div><input type='hidden' name='item' value='1'/>Item 1</div>
```

```
    <div><input type='hidden' name='item' value='2'/>Item 2</div>
    <div><input type='hidden' name='item' value='2'/>Item 3</div>
</form>
```

With Sortable, as with most javascript libraries, you need to initialize content at some point.

In jquery you might do this like so:

```
$(document).ready(function() {
    var sortables = document.body.querySelectorAll(".sortable");
    for (var i = 0; i < sortables.length; i++) {
        var sortable = sortables[i];
        new Sortable(sortable, {
            animation: 150,
            ghostClass: 'blue-background-class'
        });
    }
});
```

In htmx, you would instead use the `htmx.onLoad` function, and you would select only from the newly loaded content, rather than the entire document:

```
htmx.onLoad(function(content) {
    var sortables = content.querySelectorAll(".sortable");
    for (var i = 0; i < sortables.length; i++) {
        var sortable = sortables[i];
        new Sortable(sortable, {
            animation: 150,
            ghostClass: 'blue-background-class'
        });
    }
})
```

This will ensure that as new content is added to the DOM by htmx, sortable elements are properly initialized.

If javascript adds content to the DOM that has htmx attributes on it, you need to make sure that this content is initialized with the `htmx.process()` function.

For example, if you were to fetch some data and put it into a div using the `fetch` API, and that HTML had htmx attributes in it, you would need to add a call to `htmx.process()` like this:

```
let myDiv = document.getElementById('my-div')
fetch('http://example.com/movies.json')
    .then(response => response.text())
    .then(data => { myDiv.innerHTML = data; htmx.process(myDiv); }
```

Some 3rd party libraries create content from HTML template elements. For instance, Alpine JS uses the `x-if` attribute on templates to add content conditionally. Such templates are not initially part of the DOM and, if they contain htmx attributes, will need a call to `htmx.process()` after they are loaded. The following example uses Alpine's `$watch` function to look for a change of value that would trigger conditional content:

```
<div x-data="{show_new: false}"
    x-init="$watch('show_new', value => {
        if (show_new) {
            htmx.process(document.querySelector('#new_content'))
        }
    })">
    <button @click = "show_new = !show_new">Toggle New Content</but
    <template x-if="show_new">
        <div id="new_content">
            <a hx-get="/server/newstuff" href="#">New Clickable</a>
        </div>
```

```
    </template>
</div>
```

Security

htmx allows you to define logic directly in your DOM. This has a number of advantages, the largest being Locality of Behavior making your system more coherent.

One concern with this approach, however, is security. This is especially the case if you are injecting user-created content into your site without any sort of HTML escaping discipline.

You should, of course, escape all 3rd party untrusted content that is injected into your site to prevent, among other issues, XSS attacks. Attributes starting with `hx-` and `data-hx`, as well as inline `<script>` tags should be filtered.

It is important to understand that htmx does *not* require inline scripts or `eval()` for most of its features. You (or your security team) may use a CSP that intentionally disallows inline scripts and the use of `eval()`. This, however, will have *no effect* on htmx functionality, which will still be able to execute JavaScript code placed in htmx attributes and may be a security concern. With that said, if your site relies on inline scripts that you do wish to allow and have a CSP in place, you may need to define htmx.config.inlineScriptNonce--however, HTMX will add this nonce to *all* inline script tags it encounters, meaning a nonce-based CSP will no longer be effective for HTMX-loaded content.

To address this, if you don't want a particular part of the DOM to allow for htmx functionality, you can place the `hx-disable` or `data-hx-disable` attribute on the enclosing element of that area.

This will prevent htmx from executing within that area in the DOM:

```
<div hx-disable>
    <%= user_content %>
</div>
```

This approach allows you to enjoy the benefits of Locality of Behavior while still providing additional safety if your HTML-escaping discipline fails.

Configuring htmx

Htmx has some configuration options that can be accessed either programatically or declaratively. They are listed below:

Config Variable	Info
`htmx.config.historyEnabled`	defaults to `true`, really only useful for testing
`htmx.config.historyCacheSize`	defaults to 10
`htmx.config.refreshOnHistoryMiss`	defaults to `false`, if set to `true` htmx will issue a full page refresh on history misses rather than use an AJAX request
`htmx.config.defaultSwapStyle`	defaults to `innerHTML`
`htmx.config.defaultSwapDelay`	defaults to 0
`htmx.config.defaultSettleDelay`	defaults to 20
`htmx.config.includeIndicatorStyles`	defaults to `true` (determines if the indicator styles are loaded)
`htmx.config.indicatorClass`	defaults to `htmx-indicator`
`htmx.config.requestClass`	defaults to `htmx-request`
`htmx.config.addedClass`	defaults to `htmx-added`

Config Variable	Info
`htmx.config.settlingClass`	defaults to `htmx-settling`
`htmx.config.swappingClass`	defaults to `htmx-swapping`
`htmx.config.allowEval`	defaults to `true`
`htmx.config.inlineScriptNonce`	default to '', no nonce will be added to inline scripts
`htmx.config.useTemplateFragments`	defaults to `false` , HTML template tags for parsing content from the server (not IE11 compatible!)
`htmx.config.wsReconnectDelay`	defaults to `full-jitter`
`htmx.config.disableSelector`	defaults to `[disable-htmx], [data-disable-htmx]` , htmx will not process elements with this attribute on it or a parent
`htmx.config.timeout`	defaults to 0 in milliseconds
`htmx.config.defaultFocusScroll`	if the focused element should be scrolled into view, defaults to false and can be overriden using the focus-scroll swap modifier.

You can set them directly in javascript, or you can use a `meta` tag:

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

Conclusion

And that's it!

Have fun with htmx! You can accomplish quite a bit without writing a lot of code!

haiku

javascript fatigue:
longing for a hypertext
already in hand

docs
reference
examples
talk
sponsor