HTMX Course

Flavio Copes

Updated August 3, 2026

Contents

Learn HTMX by extending HTML with requests, targets, swaps, triggers, forms, indicators, errors, history, and progressive enhancement.

What you'll learn: Build a progressively enhanced server-rendered interface that updates useful HTML fragments without a client-side application framework.

Hypermedia foundations

Start from HTML, HTTP, server routes, and progressive enhancement.

What HTMX adds to HTML

See HTMX as a small extension to HTML that lets elements issue HTTP requests and use returned HTML.

HTML already has a useful application model: a link or form sends an HTTP request, and the server returns the next document. HTMX extends that model instead of replacing it.

It lets you answer five questions in markup:

```
<button
  hx-post="/tasks/42/complete"
  hx-trigger="click"
  hx-target="closest li"
  hx-swap="outerHTML">
  Complete
</button>
```

- which event starts the interaction?
- which HTTP method and URL should be used?
- which element receives the response?
- how is the returned HTML placed there?

The server handles the request and returns HTML representing the new task state. HTMX swaps that fragment into the list. There is no JSON response followed by a second template in the browser.

This works especially well when the server already owns validation, authorization, and rendering. It is less suitable when a feature must manipulate large amounts of local state without contacting the server.

Keep this cycle in mind for the whole course: **event → HTTP request → HTML response → DOM swap**. Every HTMX attribute changes one part of that cycle.

Think in hypermedia

Model an interaction as a server route that returns the next useful HTML representation.

A hypermedia response contains both information and the controls for possible next actions. A task row can show its current state and include the form that completes it:

```
<li id="task-42">
  <span>Send invoice</span>
  <form action="/tasks/42/complete" method="post">
    <button>Complete</button>
  </form>
</li>
```

After the form runs, the server returns the next representation of that row. If the task is complete, the new HTML may remove the button entirely. The client does not need a separate rule saying when that action is valid.

When designing an interaction, write down four parts:

1. the element and event that express the user's intent
2. the HTTP route that performs or reads the operation
3. the HTML representation returned after the operation
4. the page region that representation replaces

This keeps application state and rendering decisions on the server. It also makes responses easy to inspect: the Network panel shows the exact HTML the browser was asked to use.

The tradeoff is more server round trips. Choose hypermedia when the server is the natural source of truth, not when every keystroke must update a complex offline model.

Add HTMX to a page

Load HTMX from a pinned script or installed package and verify the actual file is delivered before adding attributes.

For a page without a JavaScript build step, load the current pinned HTMX 2 distribution:

```
<script
  src="https://cdn.jsdelivr.net/npm/htmx.org@2.0.10/dist/htmx.min.js"
  integrity="sha384-H5SrcfygHmAuTDZphMHqBJLc3FhssKjG7w/CeCpFReSfwBWDTKpkzPP8c+cLsK+V"
  crossorigin="anonymous">
</script>
```

Pinning makes the deployed bytes part of your release. The integrity hash lets the browser reject a CDN response that does not match those bytes. When you upgrade, update both the version and hash from the official documentation.

In a project with an asset pipeline, install `htmx.org@2.0.10` and import the appropriate distribution entry. This lets your lockfile and dependency update process control the version.

Before adding attributes, open the browser Network panel and confirm the script returns `200`. Then run `htmx.version` in the console. If it is undefined, fix loading first; changing `hx-*` markup cannot

repair a missing library.

Avoid an unversioned CDN URL. It can change application behavior without a code change or deployment from you.

Make the first request

Turn a button click into a GET request and replace the button contents with the returned HTML.

With the library loaded, one attribute is enough for a complete round trip. Start with one request attribute:

```
<button hx-get="/hello">Load greeting</button>
```

`hx-get` declares two things at once: which HTTP method to use and which URL to request. When the button is clicked, HTMX issues the request in the background instead of navigating the page. A click sends this ordinary HTTP request:

```
GET /hello
HX-Request: true
```

There is nothing exotic in it. Any server that can answer a browser can answer this. The `HX-Request: true` header is the only hint that HTMX sent it, and the server can ignore that header entirely for now.

Make the server return a fragment, not JSON:

```
<strong>Hello from the server</strong>
```

This is the core habit to build. In a JSON-driven application the server returns data and the client renders it. Here the server returns the finished HTML, and HTMX's job ends at placing it in the page.

With no other attributes, the button is both the trigger and target. HTMX uses `innerHTML`, so it keeps the `<button>` and replaces its children:

```
<button hx-get="/hello">
  <strong>Hello from the server</strong>
</button>
```

The server is responsible for the response content. HTMX does not infer a template from data and does not know what a greeting means. If your endpoint returns JSON out of habit, HTMX still swaps it in, and you will see raw `{"message": ...}` text inside the button. That symptom means the response type is wrong, not the attribute.

Try the exchange with DevTools open. Confirm the click creates one `GET` on the Network panel, inspect the `HX-Request` header, read the response as HTML, and compare it with the resulting DOM. That evidence gives you the complete loop: event, request, response, target, and swap. Every HTMX feature in this course is a refinement of one of those five steps.

Return a full page or a fragment

Use ordinary navigation for complete documents and return focused fragments when an HTMX target needs only part of the page.

A public URL should return a complete document when opened directly. An HTMX target usually needs only the relevant fragment.

One route can support both representations:

```
GET /tasks

if HX-History-Restore-Request is true or HX-Request is absent:
  render the full page containing that fragment
else:
  render the task-list fragment
```

The fragment might be:

```
<ul id="task-list">
  <li>Send invoice</li>
</ul>
```

The full response includes the document shell, title, navigation, and the same list template. Reuse the list renderer so the two paths cannot drift into different markup.

There is one history detail: a request carrying `HX-History-Restore-Request: true` follows a cache miss and expects enough HTML to restore the page, normally a full document. Do not treat every `HX-Request` as an identical fragment request.

These headers describe rendering context, not identity. Any client can forge them.

Test `/tasks` three ways: direct navigation, an HTMX interaction, and browser back after clearing the history cache. Each response should contain the representation its consumer expects.

Start from progressive enhancement

Build core navigation and submission from real links and forms before improving it with asynchronous swaps.

Progressive enhancement means building the working baseline first and layering improvements on top. For HTMX that ordering is natural, because every `hx-*` attribute sits on an element that can already do the job by itself.

Start with an interaction the browser already understands:

```
<form action="/search" method="get">
  <label>
    Search
    <input type="search" name="q">
  </label>
  <button>Search</button>
</form>
```

Without JavaScript, the browser navigates to `/search?q=...` and the server returns a complete results page. That URL can be refreshed, bookmarked, and shared. Nothing about this form knows HTMX exists.

Now enhance the same form:

```
<form action="/search" method="get"
  hx-get="/search"
  hx-target="#results"
  hx-push-url="true">
```

```
  <!-- the same controls -->
</form>
```

When the script has loaded, HTMX intercepts the submission, updates only `#results`, and records the meaningful search URL in the address bar. When it has not loaded, the browser sees an ordinary form and follows `action` and `method`. The `action`, `method`, named control, and submit button still define the fallback, which is why they stay in the markup instead of being replaced by the `hx-*` attributes.

The common way to break this is skipping the baseline. A form with only `hx-get` and no `action` does nothing without the script; there is no fallback to fall back to. If you spot elements like that in a review, the enhancement order went backwards.

Who hits the fallback in practice? Visitors on a flaky connection where the script has not arrived yet, browsers with scripts blocked, and crawlers that read your HTML without running JavaScript. It is a real audience, not a hypothetical one.

Progressive enhancement does not require both experiences to look identical. It requires the essential task to remain understandable and usable when the script fails, is blocked, or has not loaded yet.

Disable JavaScript and complete the flow. If you cannot, fix the HTML and server routes before adding more HTMX behavior.

The server remains the security boundary

Apply authentication, authorization, validation, CSRF protection, and output escaping to every HTMX endpoint.

HTMX changes how the browser initiates a request. It does not change the server's security responsibilities.

An attacker can copy this request without clicking your button:

```
<button hx-delete="/tasks/42">Delete</button>
```

The delete route must independently:

1. authenticate the current user
2. authorize that user for task `42`
3. verify CSRF protection for the state-changing request
4. validate every submitted value
5. perform the operation safely
6. escape untrusted values in the returned HTML

Hiding a button is a user-interface decision, not authorization. `HX-Request: true`, hidden inputs, `hx-vals`, and IDs in the DOM are all controlled by the client.

Return an accurate status code and safe error fragment. A `403` may say the action is unavailable, but it must not reveal another user's private task or an internal stack trace.

As an exercise, send the same request from `curl` without your page. If the server trusts an HTMX attribute or header that the browser normally supplied, the route has a security bug.

Check your understanding: hypermedia foundations

Check what HTMX adds, HTML responses, server state, progressive enhancement, request headers, inserted content, and security boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Requests and responses

Issue HTTP requests and return the HTML each interaction needs.

Use the request method attributes

Choose hx-get, hx-post, hx-put, hx-patch, or hx-delete to match the meaning of the server operation.

Each request attribute takes a URL:

```
<button hx-post="/tasks">Create task</button>
<button hx-patch="/tasks/42">Complete task</button>
<button hx-delete="/tasks/42">Delete task</button>
```

Use the HTTP method to describe the operation:

- `hx-get` retrieves a representation without changing server state
- `hx-post` commonly creates a resource or performs a command
- `hx-put` replaces a resource representation
- `hx-patch` applies a partial change
- `hx-delete` removes a resource

The attribute only declares a request. The server still needs a matching route, authentication, authorization, validation, and CSRF protection for state-changing methods.

HTMX 2 sends values for `DELETE` in URL parameters by default, aligning it with the current specification behavior. Do not assume the request body format from an older HTMX version; inspect the actual request and configure your server parser accordingly.

GET must remain safe because browsers, crawlers, caches, and prefetchers may repeat it. A route that deletes on `hx-get` is dangerous even if the current button is the only visible way to call it.

Open the Network panel and verify method, URL, and parameter location for each example.

Use GET for safe retrieval. Use a state-changing method for create, update, or delete actions.

Send request parameters

Use named controls and ordinary form encoding so the server receives the values it expects.

How do values reach the server in an HTMX request? The answer is reassuring: HTMX follows normal form rules. If you know how a plain HTML form submits, you already know what HTMX sends.

A form sends its **successful named controls** — the controls that have a `name` and are in a submittable state:

```
<form action="/tasks" method="post" hx-post="/tasks">
  <label>
    Task
    <input name="title" required>
  </label>

  <label>
    <input type="checkbox" name="urgent" value="yes">
    Urgent
  </label>

  <button name="intent" value="create">Add task</button>
</form>
```

The server receives `title`, receives `urgent=yes` only when checked, and receives the clicked submit button's `intent`. Submitting with the task ”Buy milk” and the checkbox ticked produces a body like this:

```
title=Buy+milk&urgent=yes&intent=create
```

An unnamed control is not submitted, no matter what the user typed into it. A disabled control is not submitted either. Both rules come from HTML, not from HTMX, and both produce the same silent symptom: a value that looks fine on screen and never arrives at the server. When a field is mysteriously missing from a request, check its `name` attribute and its `disabled` state before anything else.

The method decides where the values travel. A GET encodes values in the query string, so they appear in the URL. A POST normally uses the request body according to the form encoding. The server must parse repeated names correctly for multi-select controls and checkbox groups, where `tag=a&tag=b` is two values for one field, not a mistake.

Do not infer the payload from what appears on screen. Inspect the Network panel's query string or form data, then log only safe field names on the server while debugging. Client values remain untrusted even when browser validation ran first: anyone can send any parameters to your endpoint without using your form at all.

Add a `name` to one control and remove it from another, then submit and compare the two requests in the Network panel.

Include extra values

Add values from another control or a small explicit value map without duplicating the entire form.

`hx-include` adds successful controls outside the normal request context. A toolbar button can include a separate search input:

```
<input id="task-filter" name="q" type="search">

<button
  hx-get="/tasks"
  hx-include="#task-filter"
  hx-target="#task-list">
  Apply filter
```

```
</button>
```

Use a stable selector and verify that the selected element contains named controls. Relative selectors are resolved from the element that triggers the request, which can surprise you when `hx-include` is inherited from a parent.

`hx-vals` adds a JSON-shaped set of values:

```
<button hx-delete="/tasks/42" hx-vals='{"source":"list"}'>Delete</button>
```

Prefer ordinary named controls when the value is part of a form a person can understand. Use `hx-vals` for a small piece of request context, not as a hidden client-side data model.

Both mechanisms send client-controlled values. `source=list` can help analytics or rendering, but it cannot prove where the request came from and must never grant permission.

Inspect the outgoing request and confirm each included value appears exactly once. Duplicate field names may be intentional arrays or an accidental collision your server parses unpredictably.

Inspect HTMX request headers

Use HX-Request and related headers as rendering context without treating them as proof of identity or permission.

HTMX sends request context in `HX-*` headers. Open the Network panel and look for:

- `HX-Request: true`
- `HX-Trigger`: the triggering element's ID, when it has one
- `HX-Trigger-Name`: its name, when present
- `HX-Target`: the target's ID, when present
- `HX-Current-URL`: the browser's current URL
- `HX-Boosted: true` for a boosted request
- `HX-History-Restore-Request: true` after a history cache miss

The server can use these as rendering context. `HX-Request` often selects fragment versus full-page output. `HX-History-Restore-Request` tells the route that history restoration needs enough HTML to reconstruct the page.

Do not make correctness depend on optional IDs. If `HX-Target` is absent, the route should still have a sensible representation.

Every one of these headers can be forged by a script or command-line client. They do not prove identity, permission, CSRF validity, or that the request originated from your markup.

Copy the request as `curl`, change `HX-Target`, and send it again. The response may render differently; authorization must not.

Return the updated HTML

Let the server render the changed state and return the exact fragment the target needs.

Return the representation the target needs after the server has decided the new state.

For a task list target, a successful create can return the complete list:

```
<ul id="task-list">
  <li>Send invoice</li>
  <li>Book train</li>
```

```
</ul>
```

Returning only the new row is smaller, but returning the full list also keeps sorting, totals, empty states, and permissions consistent. Choose the smallest fragment that contains the complete changed state.

Validation is also HTML. If a form targets itself with `outerHTML`, return the form with safe values and field-specific messages:

```
<form id="new-task" action="/tasks" method="post" hx-post="/tasks"
  hx-target="this" hx-swap="outerHTML">
  <p class="error">Enter a title.</p>
  <!-- labeled controls with the safe submitted values -->
</form>
```

Render the fragment through the same server templates used by full pages. Do not duplicate escaping and presentation rules in client JavaScript.

Test the endpoint directly. Its response should be valid in the target's HTML context and should never expose private fields merely because only part of the page is returned.

Use status codes deliberately

Distinguish success, validation failure, missing resources, and server failure so HTMX and your error handling can react correctly.

HTTP status and swap behavior are related but separate decisions.

Successful `2xx` responses normally swap their HTML. `204 No Content` is a deliberate exception: HTMX performs no content swap. It fits an operation whose visible result arrives through an out-of-band update or event, but do not use it when the user needs confirmation.

Error responses such as `404`, `422`, and `500` are not swapped by default. HTMX triggers `htmx:responseError`, and `htmx:beforeSwap` receives `shouldSwap: false`.

Keep accurate status codes. Do not return `200` for every failure merely to force markup into the page. Instead, explicitly allow a safe validation fragment:

```
document.addEventListener("htmx:beforeSwap", event => {
  if (event.detail.xhr.status === 422) {
    event.detail.shouldSwap = true
    event.detail.isError = false
  }
})
```

Use this only when the `422` body is designed for the target. A generic proxy error page or stack trace must not be swapped into a form.

Force each status in development and inspect the event, response body, target, and final DOM.

Control the result with response headers

Use an HX- response header when the server knows the correct target, swap, history update, event, refresh, or redirect.

Response headers let the server refine what should happen after it knows the result.

A failed create can send its form to an error region:

```
HTTP/1.1 422 Unprocessable Content
HX-Retarget: #task-form
HX-Reswap: outerHTML
Content-Type: text/html
```

Other useful controls include:

- `HX-Push-Url` and `HX-Replace-Url` for history
- `HX-Trigger` for a named browser event
- `HX-Refresh: true` for a full refresh
- `HX-Redirect` for a full client redirect
- `HX-Location` for an HTMX-style navigation

Use headers when the server result determines the behavior. Keep stable defaults in markup when the behavior belongs to the component regardless of outcome.

HTMX-specific response headers are not processed on an ordinary `3xx` response because the browser follows the redirect before HTMX sees the original headers. Return an appropriate non-3xx response when using `HX-Redirect` or `HX-Location`.

Do not turn headers into an invisible application protocol. Inspect and document the small set each endpoint may return.

Debug an HTMX exchange

Use the browser Network panel to verify the trigger, method, URL, parameters, headers, status, response body, target, and swap.

Debug one exchange from left to right instead of changing several attributes at once.

1. Confirm HTMX loaded. The script request should succeed and `htmx.version` should exist.
2. Trigger the interaction with the Network panel open. If no request appears, inspect the natural event and `hx-trigger`.
3. Check the method, final URL, query string or body, and `HX-*` request headers.
4. Check the response status, `Content-Type`, HTMX response headers, and raw HTML body.
5. Inspect the resolved target and swap strategy.
6. Check the console for `htmx:targetError`, `htmx:swapError`, or browser parsing errors.

Classify the failure before fixing it:

- no request: loading or trigger problem
- wrong request: method, URL, or parameter problem
- wrong response: server route, validation, or template problem
- correct response but wrong DOM: target, swap, or invalid HTML context
- correct DOM but poor experience: focus, indicator, history, or accessibility problem

Reproduce the request outside HTMX with `curl` when necessary. If the endpoint itself returns the wrong fragment, no client attribute can repair it.

Check your understanding: requests and responses

Check methods, form values, query parameters, fragments, 204 responses, redirects, validation markup, and server protections.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Targets and swaps

Choose which DOM region changes and how new HTML replaces it.

Understand the default target

Know that HTMX normally swaps the response into the element that initiated the request.

When no `hx-target` is present, the element that issued the request is the target:

```
<button hx-get="/availability/42">
  Check availability
</button>
```

The default swap is `innerHTML`. If the server returns `<strong>In stock</strong>`, the result is:

```
<button hx-get="/availability/42">
  <strong>In stock</strong>
</button>
```

The button remains, including its `hx-get`, so it can be clicked again. This is useful when a control is meant to update its own label or contents.

It is wrong when the response represents another region. A button that loads an account panel should target the panel, not place a `<section>` inside the button.

Before adding `hx-target`, predict the exact resulting DOM under the default. If that markup is invalid or surprising, choose an explicit target. Inspect the Elements panel after the swap rather than relying on the visual result—the browser may repair invalid nesting in unexpected ways.

Choose a target

Point the response at a stable page region with a CSS selector or a relative target.

A search input can update a results container:

```
<input name="q" hx-get="/search" hx-target="#results">
<div id="results"></div>
```

`hx-target` accepts a CSS selector and is inherited, so a stable container can define the target for several controls.

Choose the smallest region that contains the complete changed state. If a search changes the result rows, count, and empty message, target a wrapper containing all three—not only the `<ul>`.

A target that is too broad replaces unrelated state. It may reset focus, scroll position, media playback, or client widgets. A target that is too narrow leaves stale totals and actions beside fresh results.

Give major target regions stable IDs because they are easy to inspect, address from response headers, and preserve across full-page and fragment templates. Do not reuse the same ID elsewhere in the document.

Exercise both response shapes: return three results, then return none. If every piece of search state becomes correct from one fragment, the boundary is coherent. If client code must clean up neighboring elements afterward, widen the target or use one deliberate out-of-band update.

Use relative targets

Build reusable controls with this, closest, find, next, and previous instead of unique ids for every row.

Relative targets let repeated components describe their local structure without generating a unique global ID for every row:

```
<li>
  <span>Send invoice</span>
  <button
    hx-delete="/tasks/42"
    hx-target="closest li"
    hx-swap="delete">
    Delete
  </button>
</li>
```

HTMX supports several extended selectors:

- `this` targets the element carrying the attribute
- `closest <selector>` walks through ancestors, including the element itself
- `find <selector>` finds the first matching descendant
- `next` and `previous` choose a sibling
- `next <selector>` and `previous <selector>` scan forward or backward

The selector is resolved from the triggering element, including when an inherited attribute was declared on a parent. That detail matters for `find`: the trigger's descendants may differ from the parent's descendants.

Use relative selectors for a stable component relationship such as “this row” or “the next error message.” Use an ID when the target is a page-level region unrelated to the trigger's local tree.

Duplicate the row and confirm both buttons update only their own copy.

Choose innerHTML or outerHTML

Decide whether the target container remains or the response replaces the target element itself.

`innerHTML` keeps the target element and replaces its children:

```
<ul id="task-list" hx-get="/tasks" hx-swap="innerHTML">
  <!-- the response should contain list items -->
</ul>
```

This is the default. Attributes and identity on `#task-list` remain in place.

`outerHTML` replaces the target itself:

```
<article id="task-42"
  hx-get="/tasks/42/edit"
  hx-swap="outerHTML">
```

```
  ...
</article>
```

The response must include a complete valid replacement `<article>`, including any ID and HTMX attributes needed for later interactions. Returning only its children would remove the component boundary.

HTML parsing context matters. A `<tr>` response belongs inside a table structure, and browsers may move or discard invalid standalone elements. Use `<template>` wrapping where the out-of-band rules require it, and inspect the resulting DOM.

HTMX cannot replace `<body>` with `outerHTML`; it falls back to `innerHTML`. For page-level navigation, use boosted links or a stable child target instead of depending on body replacement semantics.

Insert before or after existing content

Append, prepend, or place a returned fragment next to the target without rerendering the entire list.

Adjacent swaps use the same positions as `insertAdjacentHTML`:

- `beforebegin`: before the target itself
- `afterbegin`: inside the target, before its first child
- `beforeend`: inside the target, after its last child
- `afterend`: after the target itself

To append a newly created task:

```
<form action="/tasks" method="post"
  hx-post="/tasks"
  hx-target="#task-list"
  hx-swap="beforeend">
  ...
</form>

<ul id="task-list"></ul>
```

The server returns one valid `<li>`. The list remains and the new row is inserted at its end.

Appending is correct only when the new row is the complete change. If creation also changes sorting, a total, pagination, or an empty-state message, returning the whole list region is simpler and less prone to stale UI.

Also consider concurrent requests. Two responses may arrive in a different order than they were sent. If order matters, synchronize the requests or let the server return the complete authoritative ordering.

Delete a target or skip the main swap

Use delete for a removed element and none when the request has another effect but no normal target representation.

Use `delete` when a successful server operation means the current target no longer has a representation:

```
<button
```

```
  hx-delete="/tasks/42"
  hx-target="closest li"
  hx-swap="delete">
  Delete
</button>
```

After a successful response, HTMX removes the target regardless of the response body. The server must delete the stored task first and return an error when it cannot. Removing DOM optimistically does not authorize or persist anything.

Use `none` when the normal target should not receive response content:

```
<button hx-post="/analytics/dismiss" hx-swap="none">
  Dismiss tip
</button>
```

Response headers, events, and out-of-band fragments are still processed with `none`. That makes it useful for a request whose visible update is elsewhere.

Do not use `none` to hide a missing success state. If a person needs to know whether an operation worked, return or trigger an accessible confirmation. Force a server error and verify that the target remains recoverable.

Update another region out of band

Return a secondary fragment that updates a stable element outside the normal target.

An out-of-band swap lets one response update a second stable region in addition to the normal target.

Suppose adding a task replaces the form normally but must also update the count. The response can contain both fragments:

```
<form id="new-task" hx-post="/tasks" hx-swap="outerHTML">
  <!-- cleared form -->
</form>

<span id="task-count" hx-swap-oob="true">3 tasks</span>
```

HTMX extracts the element marked `hx-swap-oob`, finds the existing element with the same ID, and replaces it outside the main target. The OOB element does not remain inside the form response.

The attribute can also specify another swap style and selector. Keep the default `true` form when replacing a stable element is enough.

Some elements need valid parsing context. For table rows, use the wrapping pattern documented for OOB swaps, often with `<template>`, so the browser does not discard the fragment before HTMX processes it.

Use OOB swaps for a small number of consequences from the same server action, such as a count and flash message. If one response updates many unrelated areas, your target boundary or page representation is probably too fragmented.

Understand swap and settle

Use the HTMX request, swapping, added, and settling classes to make transitions explain state rather than hide slow work.

An HTMX update has observable phases:

1. the request starts and `htmx-request` marks the trigger or indicator
2. the response arrives and `htmx-swapping` marks the target
3. new content receives `htmx-added`
4. the target receives `htmx-settling` while attributes and transitions settle
5. temporary classes are removed

The default swap delay is `0ms`; the default settle delay is `20ms`. Change them only to coordinate a meaningful transition:

```
<div hx-get="/notice" hx-swap="innerHTML swap:100ms settle:200ms">
  Load notice
</div>
```

Events follow the same boundary: `htmx:afterSwap` runs after insertion, while `htmx:afterSettle` runs after settling completes.

Use CSS to explain state, not disguise latency. A short fade can show that content changed; it cannot replace a "Saving…" indicator or an error message. Respect `prefers-reduced-motion` and keep the final DOM correct when transitions are disabled.

Slow the request in development and watch the classes in the Elements panel. This turns timing problems into visible lifecycle evidence.

Check your understanding: targets and swaps

Check default and relative targets, inner and outer replacement, adjacent insertion, delete, none, out-of-band updates, and valid fragment context.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Triggers, forms, and feedback

Control timing, submit values, and explain in-flight work.

Start from natural triggers

Use the element event people already expect before customizing request timing.

HTMX chooses a default trigger from the element:

- forms trigger on `submit`
- inputs, textareas, and selects trigger on `change`
- most other elements trigger on `click`

That makes a form work with browser conventions:

```
<form action="/tasks" method="post" hx-post="/tasks">
```

```
  <label>Task <input name="title"></label>
  <button>Add task</button>
</form>
```

Pressing Enter in the field and activating the button both submit the form. HTMX also respects native constraint validation before issuing the request.

Start with the event people already expect. A `click` handler on a `<div>` loses button semantics and keyboard behavior. A form listening only to a button click can miss keyboard submission and alternate submit buttons.

Use `hx-trigger` when timing or a different event improves the feature, such as debounced search. Test the interaction with a keyboard and inspect the Network panel to confirm one user action produces one intended request.

Choose a custom trigger

Request on load, visibility, input, a custom event, or a timed interval when the feature genuinely requires it.

Natural triggers cover clicks, changes, and submits. Some features need a request without any direct user action on the element: content that loads when scrolled into view, a status that refreshes on a timer, or a request started by another part of your code. `hx-trigger` handles all of these.

```
<section hx-get="/recommendations" hx-trigger="revealed">
  Loading recommendations…
</section>
```

Use `load` when a fragment is required immediately after initialization. Use `revealed` when scrolling the window should lazy-load it. For an element inside an overflow container, use `intersect` so Intersection Observer detects visibility correctly. The placeholder text matters here: it is what people see until the trigger fires, so make it honest about what is coming.

Polling uses a timing declaration:

```
<div hx-get="/jobs/42/status" hx-trigger="every 5s">
  Processing…
</div>
```

The server can return status `286` to stop HTMX polling when the job reaches a terminal state. Without that stop condition, finished jobs keep generating a request every five seconds for as long as the tab stays open. Still avoid polling when a user action, normal refresh, SSE extension, or WebSocket extension better matches the update model.

The most flexible option is a custom event, which keeps the HTTP behavior visible in markup:

```
<div hx-get="/notifications"
  hx-trigger="notificationsChanged from:body"></div>
```

Another script or an `HX-Trigger` response header can emit `notificationsChanged`. Dispatching it from JavaScript is one line:

```
document.body.dispatchEvent(new Event('notificationsChanged'))
```

The `from:body` modifier tells HTMX to listen on the body instead of the element itself. This solves cases where no element event fits. A `<select>` where one specific option must load content is a

classic example: options fire no events of their own, so a change handler dispatches a custom event on the body and the HTMX element reacts to it. The event connects features; the element still declares the route and response target.

When a custom trigger seems dead, check the wiring first: the event must be dispatched on the same element named in `from:`. Test it by dispatching the event from the DevTools console and watching the Network panel for the request.

Use trigger modifiers

Debounce input, suppress duplicate values, limit frequency, filter events, or allow a trigger only once.

An active search can use:

```
<input type="search" name="q"
  hx-get="/search"
  hx-trigger="keyup changed delay:500ms, search"
  hx-target="#results"
  hx-sync="this:replace">
```

`changed` suppresses requests when the value is identical. `delay:500ms` is a debounce: each new keyup resets the timer. The `search` event also handles clearing the native search field.

Debouncing does not cancel a request already in flight. `hx-sync="this:replace"` aborts the older search when a newer one starts, preventing a slow old response from overwriting fresh results.

Other modifiers solve different problems:

- `throttle:500ms` allows at most one request per interval
- `once` accepts only the first matching event
- `from:body` listens on another element
- `target:<selector>` filters events by their original target
- `queue:first`, `queue:last`, or `queue:all` controls events arriving during a request

Use the smallest set that expresses a real timing requirement. With Network throttling enabled, type quickly and confirm the final results always match the current input.

Enhance a real form

Keep labels, action, method, validation, submit behavior, and server response useful before adding asynchronous replacement.

Start with a complete form:

```
<form id="new-task" action="/tasks" method="post"
  hx-post="/tasks"
  hx-target="#task-list"
  hx-swap="outerHTML">
  <label>Task <input name="title" required></label>
  <button type="submit">Add</button>
</form>
```

The `action` and `method` define the normal browser submission. The server should return a complete page with either the updated list or a form containing validation errors when JavaScript is

unavailable.

For an HTMX request, success can return a complete replacement for `#task-list`. Validation needs a different target, so the server can respond with safe form HTML plus:

```
HTTP/1.1 422 Unprocessable Content
HX-Retarget: #new-task
HX-Reswap: outerHTML
```

Configure `htmx:beforeSwap` once to allow the intentional `422` fragment. Preserve safe submitted values and associate each message with its control.

The browser's `required` check improves feedback but is not server validation. Submit the endpoint directly with an empty title and confirm the same rule still runs.

Finally, disable JavaScript. Creating a valid task and correcting an invalid one should still be possible through full-page responses.

Upload a file

Use multipart form encoding and a real file input so the server receives binary data with the other form values.

File content cannot travel in the default form encoding. Uploads need `multipart/form-data`, the HTTP encoding that carries binary parts alongside ordinary fields. A real upload form declares it explicitly:

```
<form action="/attachments" method="post"
  enctype="multipart/form-data"
  hx-post="/attachments"
  hx-target="#upload-result">
  <label>
    Attachment
    <input type="file" name="attachment" accept="image/*" required>
  </label>
  <button>Upload</button>
</form>

<div id="upload-result"></div>
```

Without JavaScript, the browser posts the file to `/attachments` and navigates. With HTMX loaded, the same submission happens in the background and the server's response lands in `#upload-result`. The file arrives under the field name `attachment`, next to any other named controls in the form.

You can alternatively set `hx-encoding="multipart/form-data"` on the HTMX request context. Keep the native `enctype` when the form must work without JavaScript.

The `hx-encoding` attribute becomes essential when the upload does not flow through a form at all. In a drag-and-drop workflow, your code collects files and starts the request with `htmx.ajax()`, which cannot set the multipart content type by itself. Put `hx-encoding` on the element and pass it as the `source`:

```
htmx.ajax('POST', '/attachments', {
  values: { attachment: formData.getAll('attachment') },
  source: dropzone,
```

```
})
```

The request inherits the `source` element's attributes, including the encoding, so the files are sent as real multipart parts. Confirm it in the Network panel: the request's `Content-Type` must be `multipart/form-data` with a boundary, not `application/x-www-form-urlencoded`.

HTMX uses `XMLHttpRequest`, so `htmx:xhr:progress` can drive a progress display for large uploads. Treat it as feedback, not proof that the server accepted or stored the file.

The server carries the real safety duties. It must enforce a size limit, inspect actual content rather than trusting the filename or MIME declaration, generate a safe storage name, authorize who may attach the file, and keep uploads outside executable paths. `accept="image/*"` only filters the file chooser; it validates nothing, and anyone can send any bytes to your endpoint directly.

Test an oversized file, a renamed non-image, a lost connection, and a valid upload. Every path needs a recoverable response the person can act on.

Show request progress

Reveal a meaningful status while work is in flight and hide it again when the request ends.

Point `hx-indicator` at a meaningful status element:

```
<button hx-post="/tasks/42/complete" hx-indicator="#save-status">
  Complete
</button>

<span id="save-status" class="htmx-indicator" role="status">
  Saving…
</span>
```

While the request is active, HTMX adds `htmx-request` to the selected indicator. Its built-in indicator styles reveal `.htmx-indicator`; you can provide your own CSS if you need a different presentation.

Use text that explains the operation. A spinner alone does not tell someone whether the page is loading, saving, deleting, or stuck.

An indicator covers only the in-flight state. On success, the returned HTML should show the new task state. On failure, replace “Saving…” with a clear error and retry path rather than simply hiding it.

Throttle the connection in DevTools so the indicator remains visible long enough to inspect. Confirm it appears only for the related request and that the control is not left in a busy state after an error.

Prevent duplicate submissions

Temporarily disable the relevant control while the server processes a request and make the endpoint safe if duplicates still arrive.

Disable the relevant control while its request is in flight:

```
<form action="/orders" method="post"
  hx-post="/orders"
  hx-disabled-elt="find button">
  <!-- order fields -->
```

```
  <button type="submit">Place order</button>
</form>
```

HTMX adds the native `disabled` attribute before sending and removes it when the request finishes. `this`, `closest`, `find`, `next`, and `previous` selectors let you choose the right scope. Disable only what would repeat or conflict with the operation; freezing the whole page is rarely necessary.

This improves the interface but is not a data-integrity guarantee. A client can bypass the attribute, two tabs can submit together, and a retry may arrive after the first response was lost.

For costly operations, send an idempotency key or enforce a server-side uniqueness constraint. The endpoint should recognize the same logical operation and return its existing result instead of creating a duplicate.

Use network throttling and double-click deliberately. Then replay the request outside the page. The server—not the temporary disabled state—must keep the data correct.

Confirm a dangerous action

Add a basic client confirmation without confusing it with server authorization or recoverability.

Add a confirmation to an action whose consequence is difficult to reverse:

```
<button
  hx-delete="/tasks/42"
  hx-confirm="Permanently delete this task?"
  hx-target="closest li"
  hx-swap="delete">
  Delete
</button>
```

HTMX uses the browser's `confirm()` dialog. Canceling prevents the request; accepting continues the normal request cycle.

Confirmation is friction, not security. The delete route must still authenticate, authorize the specific task, validate CSRF protection, and behave safely when repeated. A command-line client never sees the dialog.

Use confirmation sparingly. If nearly every action asks “Are you sure?”, people learn to approve without reading. An archive action or short-lived undo often protects users better than a modal warning.

Test both choices in the Network panel: cancel should produce no request, while confirm should produce exactly one. Then force a `403` or `500` and make sure the row is not visually removed when the server rejects the action.

Check your understanding: triggers, forms, and feedback

Check natural triggers, debounce modifiers, extra values, file encoding, indicators, disabled controls, accessibility, and confirmations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

History, errors, and enhancement

Preserve navigation, recover from failures, and keep HTML usable.

Boost links and forms

Enhance ordinary navigation and submission while preserving the original HTML behavior without JavaScript.

`hx-boost` intercepts ordinary links and forms while preserving their native fallback:

```
<main hx-boost="true">
  <a href="https://flaviocopes.com/tasks">Tasks</a>

  <form action="/search" method="get">
    <input name="q" type="search">
    <button>Search</button>
  </form>
</main>
```

A boosted same-origin link sends a GET to its `href`, swaps the response into `<body>` with `innerHTML` by default, scrolls to the top, and pushes the URL into history. A boosted form uses its `action` and `method`, but does not push a URL unless you request that separately.

The server should return complete pages for boosted navigation. HTMX can update the document title and choose the body content, while direct navigation and JavaScript-disabled browsing receive the same useful response.

`hx-boost` is inherited. Set `hx-boost="false"` on a child that must keep normal navigation, such as a download or third-party flow. Only same-origin, non-anchor links are boosted.

Inspect `HX-Boosted: true` when the server needs rendering context, but never treat it as authorization. Test direct navigation, refresh, back, and forward after enabling boost on a region.

Push a URL into history

Give a substantial content change a shareable location and make back and forward navigation restore it.

Push history when an interaction creates a navigable state, not for every DOM change:

```
<a href="https://flaviocopes.com/tasks?status=complete"
  hx-get="/tasks?status=complete"
  hx-target="#task-view"
  hx-push-url="true">
  Completed tasks
</a>
```

After the swap, the address bar contains the fetched URL. HTMX snapshots the current page state before navigation so the Back button can restore it. If the snapshot is unavailable, HTMX requests the URL with `HX-History-Restore-Request: true`.

Every pushed URL must work when pasted into a new tab or refreshed. That route should return a complete document for ordinary navigation and enough page HTML for history restoration, not only the inner list fragment.

You can set `hx-push-url` to an explicit URL when the request route and public location differ. Keep that mapping rare and obvious; otherwise the Network request, browser location, and server routing become hard to reason about.

Do not push transient states such as "menu open" or "save indicator visible." Push search, filters, pagination, or selected resources that people may share and revisit. Test direct load, refresh, Back, and Forward—not only the forward click.

Keep sensitive content out of history snapshots

Understand that HTMX can store DOM snapshots locally and disable snapshotting where the page contains sensitive data.

HTMX 2 can store DOM snapshots in `localStorage` so Back and Forward restoration feels immediate. On a shared computer, that cache may outlive the visible page.

Prevent the current page state from entering that cache with:

```
<main hx-history="false">
  <!-- account or medical information -->
</main>
```

The attribute can appear anywhere in the current document or an inserted fragment. It embargoes the whole page snapshot, not only its own subtree. History navigation still works, but HTMX requests the URL from the server when it must restore that state.

The server must authenticate and authorize the restoration request exactly like direct navigation. `HX-History-Restore-Request` is rendering context, not permission.

Do not render API keys, raw tokens, unnecessary personal data, or secrets into HTML in the first place. `hx-history="false"` protects against one browser cache; it does not remove content from the DOM, screenshots, extensions, logs, or network responses.

Test by navigating away, inspecting local storage, and using Back. Confirm the sensitive markup was not snapshotted and the server still denies restoration after the session expires.

Handle HTTP errors

Show useful server failure or validation markup while keeping the affected controls recoverable.

An HTTP error means the server responded, but the result was not successful. HTMX triggers `htmx:responseError`, and it does not swap the body by default.

Treat expected validation differently from an unexpected server failure. A `422` response can contain a form designed for display, so allow it deliberately:

```
document.addEventListener("htmx:beforeSwap", event => {
  const status = event.detail.xhr.status

  if (status === 422) {
    event.detail.shouldSwap = true
    event.detail.isError = false
  }
})
```

The server can use `HX-Retarget` to send that form to a stable region. A `500` should instead

produce a generic message and retry path; never swap a framework error page or stack trace into the application.

Keep the status accurate for monitoring and debugging. Handle `401`, `403`, `404`, `409`, `422`, and `500` according to their meaning rather than collapsing every problem into `200`.

Force each relevant response in development. Verify the final DOM, focus, enabled controls, visible message, and Network status—not only that an event fired.

Handle connection errors

Distinguish a server response from a request that never completed and provide a retry path.

A connection error is different from an HTTP error: no usable server response arrived. HTMX emits `htmx:sendError`. Timeouts and aborted requests emit their own events.

Provide one shared recovery path:

```
document.addEventListener("htmx:sendError", () => {
  const status = document.querySelector("#connection-status")
  if (status) status.textContent = "Connection lost. Check your network and retry."
})
```

HTMX removes request-state classes and restores elements managed by `hx-disabled-elt` as the lifecycle ends. Preserve safe form input, replace indefinite "Saving…" text, and leave a clear retry control.

For a state-changing request, "no response" does not prove "nothing happened." The server may have committed the operation before the connection failed. Blindly retrying a purchase or create request can duplicate it.

Use idempotency keys for costly operations, or reconcile by fetching the current server state before retrying. Phrase the message honestly: "We could not confirm the result" is more accurate than "Save failed."

Test offline mode, timeout, abort, and a real `500` separately. They should not all collapse into the same diagnosis.

Use request lifecycle events sparingly

Add cross-cutting behavior around beforeRequest, afterRequest, beforeSwap, afterSwap, and afterSettle without hiding core application logic.

HTMX exposes lifecycle events around two related processes: the HTTP request and the DOM update.

The request side includes:

- `htmx:beforeRequest`, which can cancel with `preventDefault()`
- `htmx:beforeSend`, immediately before transmission and too late to cancel
- `htmx:afterRequest`, which reports the completed request as successful or failed

The response and DOM side includes:

- `htmx:beforeSwap`, which can change `shouldSwap`, target, or swap behavior
- `htmx:afterSwap`, after new DOM is inserted
- `htmx:afterSettle`, after settling finishes

Do not use `afterRequest` as a substitute for a DOM event. With swap or settle delays, the request can be complete before the visual update you care about.

Use the narrowest event that matches the concern. Focus a newly revealed error after the swap; do not wait for `afterRequest` and then guess whether insertion finished.

A global listener is appropriate for shared observability:

```
document.addEventListener("htmx:afterRequest", event => {
  console.debug("HTMX request", {
    url: event.detail.xhr.responseURL,
    successful: event.detail.successful
  })
})
```

Avoid moving domain logic into global events. Authorization, validation, and state transitions belong on the server; method, URL, target, and swap should remain visible near the HTML. Otherwise the page becomes difficult to understand without tracing hidden listeners.

Manage focus after a swap

Keep keyboard users oriented when the element they were using is replaced or an error appears elsewhere.

HTMX preserves focus for compatible input elements that keep the same stable `id` across a swap. This makes replacing a validation form less disruptive:

```
<label for="task-title">Task</label>
<input id="task-title" name="title" value="">
```

Keep that ID in both the initial and error form. Use `focus-scroll:true` on `hx-swap` only when restoring focus should also bring the field back into view; the default avoids unexpected scrolling during a long request.

Some changes require an intentional move. If validation reveals an error summary, make it programmatically focusable and focus it after the swap:

```
<div id="form-errors" tabindex="-1" role="alert">
  Fix the highlighted fields.
</div>
```

A small `htmx:afterSwap` listener can focus that element when it appears. If the swap removes the focused row, move focus to the next sensible control or the list heading.

Do not move focus for every count, status, or background update. Keyboard users need continuity, not surprise. Test create, validation failure, deletion, and history restoration using only the keyboard.

Test without JavaScript

Complete the core navigation and form flow using only the href, action, method, validation, and full server responses.

Disable JavaScript and complete the core workflow as an ordinary website.

Links need real `href` values. Forms need `action`, `method`, named controls, and submit buttons.

The server must return complete pages with preserved safe values and readable validation errors.

Native forms support GET and POST, not DELETE or PATCH. Give an enhanced destructive action a POST fallback route:

```
<form action="/tasks/42/delete" method="post"
  hx-delete="/tasks/42"
  hx-target="closest li"
  hx-swap="delete">
  <button>Delete</button>
</form>
```

The server supports both routes with the same authorization and deletion logic. Without JavaScript, the POST returns a redirect or complete page. With HTMX, the DELETE returns a fragment-compatible result.

Active search can fall back to a normal GET submit. Inline completion can fall back to a small POST form and full-page response. The enhanced experience may be faster, but it should not be the only path to the essential task.

Also simulate a failed script request, not only a deliberate browser setting. Progressive enhancement protects the period before HTMX loads as well as users who disable it.

Build a progressively enhanced task interface

Combine the complete course into a list that works as ordinary HTML and becomes faster with HTMX.

Build a task interface that remains a normal server-rendered site underneath its HTMX enhancements.

Start with full-page routes for listing, creating, completing, deleting, and filtering tasks. Use real links and forms. Put authentication, per-task authorization, CSRF protection, validation, and output escaping in shared server logic before creating fragment responses.

Then enhance one interaction at a time:

- create returns the complete list region so ordering, count, and empty state stay coherent
- validation returns the form with status `422`, preserved safe values, and field messages
- completion replaces one task row with `outerHTML`
- deletion removes its row only after a successful server response
- a status filter updates the task region and pushes a directly navigable query URL

Add a textual request indicator and disable the relevant submit control while creating. Make creation idempotent if duplicates matter. Provide a site-level connection message for `htmx:sendError`, but distinguish an unknown network result from a confirmed HTTP failure.

Keep full pages and fragments in shared templates. Handle `HX-History-Restore-Request` with enough HTML to restore the page. Add `hx-history="false"` if any task content is too sensitive for local history snapshots.

Test the finished system as a contract:

1. open every public URL directly and refresh it
2. use Back and Forward across filters
3. submit empty, duplicate, unauthorized, and valid values

4. force `422`, `403`, `500`, timeout, and offline behavior
5. complete the workflow with JavaScript disabled
6. use only the keyboard and verify focus after errors and deletion
7. inspect every request, response fragment, target, and swap

The final interface should feel faster with HTMX, while correctness remains in HTTP routes and server-rendered representations.

Check your understanding: history, errors, and enhancement

Check boosted links and forms, pushed URLs, direct navigation, history caches, HTTP and network errors, cancellation, recovery, and focus.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/htmx/.