Hypermedia In Action

3. Extending HTML As Hypermedia

This chapter covers

- The shortcomings of "plain" HTML
- How htmx addresses these shortcomings
- How to issue various HTTP requests with htmx
- History and back button support in htmx

3.1. The Shortcomings of "Plain" HTML

In the previous chapter we introduced a simple Web 1.0-style hypermedia application to manage contacts. This application supported the normal CRUD operations for contacts, as well as a simple mechanism for searching contacts. Our application was built using nothing but forms and anchor tags, the traditional tags used to interact with servers, and it exchanges hypermedia (HTML) with the server over HTTP, issuing `GET` and `POST` HTTP requests and receiving back full HTML documents in response. It is pretty simple, but it is also definitely a Hypermedia Driven Application.

Our application is robust, leverages the web's strengths and is simple to understand. So what's not to like? Well, unfortunately, our application isn't completely satisfying from either a user experience perspective, or from a technical perspective. It suffers from problems typical of this style of Web 1.0 applications.

Two obvious problems that jump out are:

- From a user experience perspective: there is a noticeable refresh when you move between pages of the application, of when you create, update or delete a contact. This is because every user interaction (link click or form submission) requires a full page refresh, with a whole new HTML document to process after each action.
- From a technical perspective, all the updates are done with the `POST` HTTP action. This is despite the fact that more logical actions HTTP request types like `PUT` and `DELETE` exist and would make far more sense for some of the operations. Somewhat ironically, since we are using pure HTML, we are unable to access the full expressive power of

HTTP!

The first point, in particular, is noticeable in Web 1.0 style applications like ours and is what is responsible for giving them the reputation for being "clunky" when compared with their more sophisticated JavaScript-based Single Page Application cousins.

Single Page Applications eliminate this clunkiness by updating a web page directly, mutating the Document Object Model (DOM), the JavaScript API to the underlying HTML page. There are a few of different styles of SPA, but, as we discussed in Chapter 1, the most common today is to tie the DOM to a JavaScript model and let an SPA framework like react *reactively* update the DOM when the JavaScript model is updated: you make a change to a JavaScript object and the web page magically updates its state to reflect the change in the model.

Recall that in this style of application communication with the server is typically done via a JSON Data API, with the application sacrificing the advantages of hypermedia in order to provide a better, smoother user experience.

Many web developers today would not even consider the hypermedia approach due to the perceived "legacy" feel of these Web 1.0 style applications.

The second, technical point may strike you as a bit pedantic, and I am the first to admit that conversations around REST and which HTTP Action is right for a given operation can become very tedious. But, nonetheless, it has to be admitted that, when using plain HTML, it is impossible to use HTTP to its full power and, therefore, it is impossible to realize the full vision of the web as a REST-ful system: a complete, stateless, resource-oriented distributed networking architecture that is flexible and resilient.

3.1.1. A Close Look At A Hyperlink

As we have been saying, it turns out that you can actually get a lot of interactivity out of the hypermedia model, if you adopt a hypermedia-oriented library like htmx. To understand conceptually how htmx allows us to better address the UX concerns of Web 1.0 style applications, let’s revisit the hyperlink/anchor tag from Chapter 1 and really drill in to each facet of it:

This simple anchor tag, when interpreted by a browser, creates a hyperlink to the Manning website:

Listing 3. 1. A Simple Hyperlink, Again

```
<a href="https://www.manning.com/">
  Manning Books
</a>
```

Breaking down exactly what this link will tell the browser to do, we have the following list:

- The browser will render the text "Manning Books" to the screen, likely with a decoration indicating it is clickable
- Then, when a user clicks on the text…
- The browser will issue an HTTP `GET` to `https://www.manning.com` and then…
- The browser will load the HTTP response into the browser window, replacing the current document

So we have four aspects of a simple hypermedia link like this, with the last three being the mechanic that distinguishes a hyperlink from "normal" text.

Let's take a moment and think about how we can generalize this fundamental hypermedia mechanic of HTML. There is no rule saying that hypermedia can *only* work this way, after all!

An initial observation is: why are anchor tags so special? Shouldn't other elements (besides forms) be able to issue HTTP requests as well? For example, shouldn't `button` elements be able to do so? It seemed silly to have to wrap a form tag around a button to make deleting contacts work in our application. Why should only anchor tags and forms be able to issue requests?

This presents our first opportunity to expand the expressiveness of HTML: we can allow *any* element to issue a request to the server.

For our next observation, let's consider the event that triggers the request to the server on our link: a click. Well, what's so special about clicking (in the case of anchors) or submitting (in the case of forms)? Those are just two of many, many events that are fired by the DOM, after all. Events like mouse down, or key up, or blur are all events you might want to use to issue an HTTP request. Why shouldn't these other events be able to trigger requests as well?

This gives us our second opportunity to expand the expressiveness of HTML: we can allow *any* event, not just a click, as in the case of our hyperlink, to trigger an HTTP request.

Getting a bit more technical in our thinking leads us to the problem we noted earlier in the chapter: plain HTML only give us access to the `GET` and `POST` actions of HTTP? HTTP *stands* for HyperText Transfer Protocol, and yet the format it was explicitly designed for, HTML, only supports two of the five developer-facing request types! You *have* to use JavaScript and issue an AJAX request to get at the other three: `DELETE`, `PUT` and `PATCH`.

Let's recall what are all of these different HTTP request types designed to represent?

- `GET` corresponds with "getting" a representation for a resource from a URL: it is a pure read, with no mutation of the resource
- `POST` submits an entity (or data) to the given resource, often creating or mutating the resource and causing a state change
- `PUT` submits an entity (or data) to the given resource for update or replacement, again likely causing a state change
- `PATCH` is similar to `PUT` but implies a partial update rather than a complete replacement of the entity
- `DELETE` deletes the given resource

These operations correspond closely to the CRUD operations we discussed in Chapter 2, and by only giving us access to two of them, HTML is presenting us with a severe and obvious technical limitation.

So here is our third opportunity to expand the expressiveness of HTML: we can allow HTML to have access to the missing three HTTP actions, `PUT`, `PATCH` and `DELETE`.

As a final observation, consider that last aspect of a hyperlink: it replaces the *entire* screen when a user clicks on it. It is this technical detail that makes for a poor user experience: it causes flashes of unstyled content, a loss of scroll state and so forth. But, again, there is no rule saying that hypermedia exchanges *must* replace the entire document.

This gives us our forth, final and perhaps most important opportunity to generalize HTML: what if we allowed the hypermedia response to replace elements *within* the current document, rather than requiring that it replace the entire document. This would make

Hypermedia Driven Applications function much more like a Single Page Application, where only part of the DOM is updated by a given user interaction or network request.

If we were to take these four opportunities to generalize HTML, we would be extending HTML far beyond its normal capabilities, and we would be doing so *entirely within* the normal, hypermedia model of the web. Note that none of the extensions involve going outside the normal exchanging-HTML-over-HTTP found in Web 1.0 applications. Rather, the all four are simply generalizations of existing functionality already found within HTML.

3.2. Extending HTML as a Hypermedia with htmx

It turns out that there are some JavaScript libraries that extends HTML in exactly this manner. This may seem somewhat ironic, given that JavaScript-based SPAs have supplanted HTML-based hypermedia applications, that JavaScript would be used in this manner. But JavaScript is simply a language for extending browser functionality on the client side, and there is no rule saying it has to be used to write SPAs. In fact, JavaScript is the perfect tool for addressing the shortcomings of HTML as a hypermedia!

One such library is htmx, which will be the focus of the next few chapters. htmx is not the only JavaScript library that takes this hypermedia-oriented approach, but it is perhaps the purest in the pursuit of extending HTML as a hypermedia. It focuses intensely on the four limitations discussed above and attempts to incrementally address each one, without introducing a significant amount of additional conceptual infrastructure for web developers.

3.2.1. Installing and Using htmx

From a practical, getting started perspective, htmx is a simple, dependency-free and stand-alone library that can be added to a web application by simply including it via a `script` tag in your `head` element

Because of this simple installation model, we can take advantage of tools like public CDNs to install the library. Below we are using the popular unpkg Content Delivery Network (CDN) to install version `1.7.0` of the library. We use an integrity hash to ensure that the delivered content matches what we expect. This SHA can be found on the htmx website. Finally, we mark the script as `crossorigin="anonymous"` so no credentials will be sent to the CDN.

Listing 3. 2. Installing htmx

```
<head>
  <script src="https://unpkg.com/htmx.org@1.7.0"
          integrity="sha384-
EzBXYPt0/T6gxNp0nuPtLkmRpmDBbjg6WmCUZRLXBBwYYmwAUxzlSGej0ARHX0Bo"
          crossorigin="anonymous"></script>

</head>
```

Believe it or not, that's all it takes to install htmx! If you are used to the extensive build systems in today's JavaScript world, this may seem impossible or insane, but this is in the spirit of the early web: you could simply include a script tag and things would just work. And it still feels like magic, even today!

Of course, you may not want to use a CDN, in which case you can download htmx to your local system and adjust the script tag to point to wherever you keep your static assets. Or, you may have one of those more sophisticated build system that automatically installs dependencies. In this case you can use the Node Package Manager (npm) name for the library: `htmx.org` and install it in the usual manner that your build system supports.

Once htmx has been installed, you can begin using it immediately.

And here we get to the funny part of htmx: unlike the vast majority of JavaScript libraries, htmx does not require you, the user, to actually write any JavaScript!

Instead, you will use *attributes* placed directly on elements in your HTML to drive more dynamic behavior. Remember: htmx is extending HTML as a hypermedia, and we want that extension to be as natural and consistent as possible with existing HTML concepts. Just as an anchor tag uses an `href` attribute to specify the URL to retrieve, and forms use an `action` attribute to specify the URL to submit the form to, htmx uses HTML *attributes* to specify the URL that an HTTP request should be issued to.

3.3. Triggering HTTP Requests

Let's look at the first feature of htmx: the ability for any element in a web page to issue HTTP requests. This is the core functionality of htmx, and it consists of five attributes that can be used to issue the five different developer-facing types of HTTP requests:

- `hx-get` - issues an HTTP `GET` request
- `hx-post` - issues an HTTP `POST` request
- `hx-put` - issues an HTTP `PUT` request
- `hx-patch` - issues an HTTP `PATCH` request
- `hx-delete` - issues an HTTP `DELETE` request

Each of these attributes, when placed on an element, tell the htmx library: "When a user clicks (or whatever) this element, issue an HTTP request of the specified type"

The values of these attributes are similar to the values of both `href` on anchors and `action` on forms: you specify the URL you wish to issue the given HTTP request type to. Typically, this is done via a server-relative path.

So, for example, if we wanted a button to issue a `GET` request to `/contacts` then we would write:

Listing 3. 3. A Simple htmx-Powered Button

```
<button hx-get="/contacts"> ❶
  Get The Contacts
</button>
```

❶ A simple button that issues an HTTP `GET` to `/contacts`

htmx will see the `hx-get` attribute on this button, and hook up some JavaScript logic to issue an HTTP `GET` AJAX request to the `/contacts` path when the user clicks on it. Very easy to understand and very consistent with the rest of HTML.

3.3.1. It's All Just HTML!

Now we get to perhaps the most important thing to understand about htmx: it expects the response to this AJAX request *to be HTML*! htmx is an extension of HTML and, just as the response to an anchor tag click or form submission is usually expected to be HTML, htmx expects the server to respond with a hypermedia, namely with HTML.

This may come as a shock to web developers who are unused to responding to an AJAX request with anything other than JSON, which is far and away the most common response format for such requests. But AJAX requests are just HTTP requests and there is no rule

saying they must be JSON! Recall again that AJAX stands for Asynchronous Javascript & XML, so JSON is already a step away from the format originally envisioned for this API: XML. htmx simply goes another direction and expects HTML.

htmx vs. "plain" HTML responses

So, we have established that htmx expects HTML responses to the HTTP requests it makes. But there is an important difference between the HTTP responses to normal anchor and form driven requests and to htmx-powered requests like the one made by this button: in the case of htmx triggered requests, responses are often only *partial* bits of HTML.

In htmx-powered interactions we are typically not replacing the entire document, so it is not necessary to transfer an entire HTML document from the server to the browser. This fact can be used to save bandwidth as well as resource loading time, since less overall content is transferred from the server to the client and since it isn't necessary to reprocess a `head` tag with style sheets,script tags, and so forth.

A simple *partial* HTML response to the button's htmx request might look like this:

Listing 3. 4. A partial HTML Response to an htmx Request

```
<ul>
  <li><a href="mailto:joe@example.com">Joe</a></li>
  <li><a href="mailto:sarah@example.com">Sarah</a></li>
  <li><a href="mailto:fred@example.com">Fred</a></li>
</ul>
```

This is just a simple unordered list of contacts with some clickable elements in it. Note that there is no opening `html` tag, no `head` tag, and so forth: it is a raw HTML list, without any decoration around it. A response in a real application might of course contain far more sophisticated HTML than a simple list, but it wouldn't need to be an entire page of HTML.

This response is perfect for htmx: it will take the returned content and swap it in to the DOM. This is fast and efficient, leveraging the existing HTML parser in the browser. And this demonstrates that htmx is staying within the hypermedia paradigm: just like in a "normal" web application, we see hypermedia being transferred to the client in a stateless and uniform manner, where the client knows nothing about the internals of the resources

being displayed.

This button just a more sophisticated component for building a Hypermedia Driven Application!

3.4. Targeting Other Elements

Now, given that htmx has issued a request and gotten back some HTML as a response, what should be done with it?

It turns out that the default htmx behavior is to simply put the returned content inside the element that triggered the request. That's obviously *not* a good thing in this situation: we will end up with a list of contacts awkwardly embedded within a button element on the page! That will look pretty silly and is obviously not what we want.

Fortunately htmx provides another attribute, `hx-target` which can be used to specify exactly where in the DOM the new content should be placed. The value of the `hx-target` attribute is a Cascading Style Sheet (CSS) *selector* that allows you to specify the element to put the new hypermedia content into

Let's add a `div` tag that encloses the button with the id `main`. We will then target this `div` with the response:

Listing 3. 5. A Simple htmx-Powered Button

```
<div id="main"> ❶

  <button hx-get="/contacts" hx-target="#main"> ❷
    Get The Contacts
  </button>

</div>
```

❶ A `div` element that wraps the button

❷ A new `hx-target` attribute that specifies the `div` as the target of the response

We have added `hx-target="#main"` to our button, where `#main` is a CSS selector that says "The thing with the ID 'main'". Note that by using CSS selectors, htmx is once again building on top of familiar and standard HTML concepts. By doing so it keeps the additional conceptual load beyond normal HTML to a minimum.

Given this new configuration, what would the HTML on the client look like after a user clicks on this button and a response has been received and processed?

It would look something like this:

Listing 3. 6. Our HTML After the htmx Request Finishes

```
<div id="main">
  <ul>
    <li><a href="mailto:joe@example.com">Joe</a></li>
    <li><a href="mailto:sarah@example.com">Sarah</a></li>
    <li><a href="mailto:fred@example.com">Fred</a></li>
  </ul>
</div>
```

The response HTML has been swapped into the `div`, replacing the button that triggered the request. This all has happened "in the background" via AJAX, without a large page refresh. Nonetheless, this is *definitely* a hypermedia interaction. It isn't as coarse-grained as a normal, full web page request coming from an anchor might be, but it certainly falls within the same conceptual model!

3.5. Swap Styles

Now, maybe we don't want to simply load the content from the *into* the div. Perhaps, for whatever reasons, we wish to *replace* the entire div with the response.

htmx provides another attribute, `hx-swap`, that allows you to specify exactly *how* the content should be swapped into the DOM. (Are you beginning to sense a pattern here?) The `hx-swap` attribute supports the following values:

- `innerHTML` - The default, replace the inner html of the target element
- `outerHTML` - Replace the entire target element with the response
- `beforebegin` - Insert the response before the target element
- `afterbegin` - Insert the response before the first child of the target element
- `beforeend` - Insert the response after the last child of the target element
- `afterend` - Insert the response after the target element

- `delete` - Deletes the target element regardless of the response
- `none` - No swap will be performed

The first two values, `innerHTML` and `outerHTML`, are taken from the standard DOM properties that allow you to replace content within an element or in place of an entire element respectively.

The next four values are taken from the `Element.insertAdjacentHTML()` DOM API, which allow you to place an element or elements around a given element in various ways.

The last two values, `delete` and `none` are specific to htmx, but should be fairly obvious for you understand.

Again, you can see that htmx tries to stay as close as possible to the existing web standards to keep your conceptual load to a minimum.

Let's consider if, rather than replacing the `innerHTML` content of the main div above, we wished to replace the *entire div* with the HTML response. To do so would require only a small change to our button:

Listing 3. 7. Replacing the Entire div

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML"> ❶
    Get The Contacts
  </button>

</div>
```

❶ The `hx-swap` attribute specifies how to swap new content in

Now, when a response is received, the *entire* div will be replaced with the hypermedia content:

Listing 3. 8. Our HTML After the htmx Request Finishes

```
<ul>
  <li><a href="mailto:joe@example.com">Joe</a></li>
  <li><a href="mailto:sarah@example.com">Sarah</a></li>
  <li><a href="mailto:fred@example.com">Fred</a></li>
</ul>
```

You can see that, with this change, the target div has been entirely removed from the DOM, and the list that was returned as the response has replaced it.

Later in the book we will see additional uses for `hx-swap`, for example when we implement infinite scrolling in our contact management application.

Note that with the `hx-get`, `hx-post`, `hx-put`, `hx-patch` and `hx-delete` attributes, we have addressed two of the shortcomings that we enumerated regarding plain HTML: we can now issue an HTTP request with *any* element (in this case we are using a button). Additionally, we can issue *any sort* of HTTP request we want, `PUT`, `PATCH` and `DELETE`, in particular.

And, with `hx-target` and `hx-swap` we have addressed a third shortcoming: the requirement that the entire page be replaced. Now we have the ability, within our hypermedia, to replace any element we want and in any manner we wish to replace it.

So, with seven relatively simple additional attributes, we have addressed most of the hypermedia shortcomings we identified earlier with HTML. Not bad!

There was one remaining shortcoming of HTML that we noted: the fact that only a `click` event (on an anchor) or a `submit` event (on a form) can trigger HTTP request. Let's look at how we can address that concern next.

3.6. Using Other Events

Thus far we have been using a button to issue a request with htmx. You have probably intuitively understood that the request will be issued when the button is clicked on since, well, that's what you do with buttons! You click on them!

And, yes, by default when an `hx-get` or another request-driving annotation from htmx is placed on a button, the request will beissued when the button is clicked.

However, htmx generalizes this notion of an event triggering a request by using, you guessed it, another attribute: `hx-trigger`. The `hx-trigger` attribute allows you to specify one or more events that will cause the element to trigger an HTTP request, overriding the default triggering event.

What is the "default triggering event" in htmx? It depends on the element type, but should be fairly intuitive to anyone familiar with HTML:

- Requests on `input`, `textarea` & `select` elements are triggered by the `change` event
- Requests on `form` elements are triggered on the `submit` event
- Requests on all other elements are triggered by the `click` event

So, lets consider if we wanted to trigger the request on our button when the mouse entered it. This is certainly not a recommended UX pattern, but let's just look at it as an example!

To do this, we would add the following attribute to our button:

Listing 3. 9. A Terrible Idea, But It Demonstrates The Concept!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="mouseenter"> ❶
    Get The Contacts
  </button>

</div>
```

❶ Issue a request… on the `mouseenter` event?

Now, whenever the mouse enters this button, a request will be triggered. Hey, we didn't say this was a *good* idea!

Let's try something a bit more realistic: let's add support for a keyboard shortcut for loading the contacts, `Ctrl-L` (for "Load"). To do this we will need to take advantage of some additional syntax that the `hx-trigger` attribute supports: event filters and additional arguments.

Event filters are a mechanism for determining if a given event should trigger a request or

not. They are applied to an event by adding square brackets after it: `someEvent[someFilter]`. The filter itself is a JavaScript expression that will be evaluated when the given event occurs. If the result is truthy, in the JavaScript sense, it will trigger the request. If not, it will not.

In the case of keyboard shortcuts, we want to catch the `keyup` event in addition to the keyup event:

Listing 3. 10. A Start

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup"> ❶
    Get The Contacts
  </button>

</div>
```

❶ A trigger with two events

Note that we have a comma separated list of events that can trigger this element, allowing us to respond to more than one potential triggering event.

There are two problems with this:

- It will trigger requests on *any* keyup event
- It will trigger requests only when a keyup occurs *within* this button (an unlikely occurrence!)

To fix the first issue, lets use a trigger filter:

Listing 3. 11. Better!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup[ctrlKey && key == 'l']"> ❶
    Get The Contacts
  </button>

</div>
```

❶ A trigger with an added filter, specifying that the control key and L must be pressed

The trigger filter in this case is `ctrlKey && key == 'l'`. This can be read as "A key up event, where the ctrlKey property is true and the key property is equal to 'l'". Note that the properties `ctrlKey` and `key` are resolved against the event rather than the global name space, so you can easily filter on the properties of a given event. You can use any expression you like for a filter, however: calling a global JavaScript function, for example, is perfectly acceptable.

OK, so this filter limits the keyups that will trigger the request to only `Ctrl-L` presses. However, we still have the problem that, as it stands, only `keyup` events *within* the button will trigger the request. If you are familiar with the JavaScript event bubbling model: events typically "bubble" up to parent elements so an event like a keyup will be triggered first on the focused element, then on it's parent, and so on, until it reaches the top level `document` that is the root of all other elements.

In this case, this is obviously not what we want! People typically aren't typing characters *within* the button, they click on buttons! Here we want to listen to the `keyup` events on the entire page, or, equivalently, on the `body` element.

To fix this, we need to take advantage of another feature that the `hx-trigger` attribute supports: the ability to listen to *other elements* for events using the `from:` modifier. The `from:`modifier, as with many other attributes and modifiers in htmx, uses a CSS selector to select the element to listen on.

We can use it like this:

Listing 3. 12. Better!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup[ctrlKey && key == 'L'] from:body">❶
    Get The Contacts
  </button>

</div>
```

❶ Listen to the event on the `body` tag

Now, in addition to clicks, our button is listening for `keyup` events on the body of the page, and should issue a request both when it is clicked on, and also whenever someone hits `Ctrl-L` within the body of the page!

A nice little keyboard shortcut! Perfect!

The `hx-trigger` attribute is more elaborate than the other htmx attributes we have looked at so far, but that is because events, in general, are used more elaborately in modern user interfaces. The default options often suffice, however, and you shouldn't need to reach for complicated trigger features too often when using htmx.

That being said, even in the more elaborate situations like the example above, where we have a keyboard shortcut, the overall feel of htmx is *declarative* rather than *imperative* and follows along closely with the standard feel and philosophy of HTML.

And hey, check it out! With this final attribute, `hx-trigger`, we have addressed *all* of the shortcomings of HTMl that we enumerated at the start of this chapter. That's a grand total of eight, count 'em, *eight* attributes that all fall squarely within the same conceptual model as normal HTML and that, by extending HTML as a hypermedia, open up world of new user interface possibilities!

3.7. Passing Request Parameters

So far we have been just looking at situation where a button makes a simple `GET` request. This is conceptually very close to what an anchor tag might do. But there is the other primary element in traditional hypermedia-based applications: forms. Forms are used to pass additional information beyond just a URL up to the server in a request. This

information is typically entered into elements within the form via the various types of input tags in HTML.

htmx allows you include this additional information in a natural way that mirrors how HTML itself works.

3.7.1. Enclosing Forms

The simplest way to pass additional input values up with a request in htmx is to enclose the input within a form tag.

Let's take our original button for retrieving contacts and repurpose it for searching contacts:

Listing 3. 13. A Simple htmx-Powered Button

```
<div id="main">

  <form> ❶
      <label for="search">Search Contacts:</label>
      <input id="search" name="q" type="search" placeholder="Search Contacts"> ❷
      <button hx-post="/contacts" hx-target="#main"> ❸
        Search The Contacts
      </button>
  </form>

</div>
```

❶ The form tag encloses the button, thereby including all values within it in the button request

❷ A new input that users will be able to enter search text into

❸ Our button has been converted to an `hx-post`

Here we have added a form tag surrounding the button along with a search input that can be used to enter a term to search the contacts with.

Now, when a user clicks on the button, the value of the input with the id `search` will be included in the request. This is by virtue of the fact that there is a form tag enclosing both the button and the input: when an htmx-driven request is triggered, htmx will look up the DOM hierarchy for an enclosing form, and, if one is found, it will include all values from within that form. (This is sometimes referred to as "serializing" the form.)

You might have noticed that the button was switched from a `GET` request to a `POST` request. This is because, by default, htmx does *not* include the closest enclosing form for `GET` requests. This is to avoid serializing forms in situations where the data is not needed and to keep URLs clean when dealing with history entries, which we discuss in the next section.

3.7.2. Including inputs

While enclosing all the inputs you want included in a request is the most common approach for including values from inputs in htmx requests, it isn't always ideal: form tags have layout consequences and cannot be placed in some places (forms, for example). So htmx provides another mechanism for including value in requests: the `hx-include` attribute which allows you to select input values that you wish to include in a request via CSS selectors.

Here is the above example reworked to include the input, dropping the form:

Listing 3. 14. A Simple htmx-Powered Button

```
<div id="main">

  <label for="search">Search Contacts:</label>
  <input id="search" name="q" type="search" placeholder="Search Contacts">
  <button hx-post="/contacts" hx-target="#main" hx-include="#search">❶
    Search The Contacts
  </button>

</div>
```

❶ `hx-include` can be used to include values directly in a request

The `hx-include` attribute takes a CSS selector value and allows you to specify exactly which values to send along with the request. This can be useful if it is difficult to colocate an element issuing a request with all the inputs that need to be submitted with it. It is also useful when you do, in fact, want to submit values with a `GET` request and overcome the default behavior of htmx with respect to `GET` requests.

3.7.3. Inline Values

A final way to include values in htmx-driven requests is to use the `hx-vals` attribute, which allows you to include static JSON-based values in the request. This can be useful if you have additional context you wish to encode during server side rendering for a request.

Here is an example:

Listing 3. 15. A Simple htmx-Powered Button

```
<button hx-get="/contacts" hx-vals='{"state":"MT"}'> ❶
  Get The Contacts In Montana
</button>
```

❶ `hx-vals`, a JSON value to include in the request

The parameter `state` the value `MT` will be included in the `GET` request, resulting in a path and parameters that looks like this: `/contacts?state=MT`. One thing to note is that we switched the `hx-vals` attribute to use single quotes around its value. This is because JSON strictly requires double quotes and, therefore, to avoid escaping we needed to use the sinqle-quote form for the attribute value.

This approach is useful when you have fixed data that you want to include in a request and you don't want to rely on something like a hidden input. You can also prefix `hx-vals` with a `js:` and pass values evaluated at the time of the request, which can be useful for including things like a dynamically maintained variable, or value from a third party javascript library.

These three mechanisms allow you to include values in your hypermedia requests with htmx in a manner that is very familiar and in keeping with the spirit of HTML.

3.8. History Support

A final piece of functionality to discuss to close out our overview of htmx is browser history. When you use normal HTML links and forms, your browser will keep track of all the pages that you have visited. You can use the back button to navigate back to a previous page and, once you have done this, you can use a forward button to go forward to the original page you were on.

This notion of history was one of the killer features of the early web. Unfortunately it turns out that history becomes tricky when you move to the Single Page Application paradigm. An AJAX request does not, by itself, register a web page in your browsers history and this is a good thing! An AJAX request may have nothing to do with the state of the web page (perhaps it is just recording some activity in the browser), so it wouldn't be appropriate to create a new history entry for the interaction.

However, there are likely to be a lot of AJAX driven interactions in a Single Page Application where it *is* appropriate to create a history entry. And JavaScript does provide an API for working with the history cache. Unfortunately the API is very difficult to work with and is often simply ignored by developers. If you have ever used a Single Page Application and accidentally clicked the back button, only to lose your entire application state and have to start over, you have seen this problem in action.

In htmx, as in Single Page Application frameworks, you often need to explicitly work with the history API. Fortunately, htmx makes it much easier to do so than most other libraries.

Consider the button we have been discussing again:

Listing 3. 16. Our trusty button

```
<button hx-get="/contacts" hx-target="#main">
  Get The Contacts
</button>
```

As it stands, if you click this button it will retrieve the content from `/contacts` and load it into the element with the id `main`, but it will *not* create a new history entry. If we wanted it to create a history entry we would add another attribute to the button, `hx-push-url`:

Listing 3. 17. Our trusty button, now with history!

```
<button hx-get="/contacts" hx-target="#main" hx-push-url="true">❶
  Get The Contacts
</button>
```

❶ `hx-push-url` will create an entry in history when the button is clicked

Now, when the button is clicked, the `/contacts` path will be put into the browser's navigation bar and a history entry will be created for it. Furthermore, if the user clicks the back button, the original content for the page will be restored, along with th original URL.

`hx-push-url` might sound a little obscure, but this is based on the JavaScript API, `history.pushState()`. This notion of "pushing" derives from the fact that history entries are modeled as a stack, and so you are "pushing" new entries onto the top of the stack of history entries.

With this (relatively) simple mechanism, htmx allows you to integrate with the back button in a way that mimics the "normal" behavior of HTML. Not bad if you look at what other javascript librarires require of you!

Drawbacks To The htmx Approach

htmx is a very pure extension to HTML, aiming to incrementally improve the language as a hypermedia in a manner that is conceptually coherent with the underlying markup language. This approach, like any technical choice, is not without tradeoffs: by staying so close to HTML, htmx does not give developers a lot of infrastructure that many might feel should be there "by default".

A good example is the concept of modal dialogs. Many web applications today make heavy use of modal dialogs, effectively in-page pop-ups that sit "on top" of the existing page. (Of course, in reality, this is an optical illusion and it is all just a web page: the web has no notion of"modals" in this regard.)

A web developer might expect htmx to provide some sort of modal dialog component out of the box, since it is, after all, a front-end library, and many front end libraries offer support for this pattern.

htmx, however, has no notion of modals. That’s not to say you can’t use modals with htmx, and we will look at how you can do so later. But htmx, like HTML itself, won’t give you an API specifically for creating modals. You would need to use a 3rd party library or roll your own modal implementation and then integrate htmx into it if you want to use modals within an htmx-based application.

This is the design tradeoff that htmx makes: it retains conceptual purity as an extension of HTML, and, in exchange, lacks some of the "batteries included" features found in other front end libraries.

As an aside, it’s worth nothing that htmx *can* be used to effectively implement a slightly different UX pattern, inline editing, which is often a good alternative to modals, and, in our opinion, is more consistent with the stateless nature of the web.

3.9. Summary

- Unfortunately, HTML has some shortcomings as a hypermedia:
  - It doesn't give you access to non-`GET` or `POST` requests
  - It requires that you update the entire page
  - It only offers limited interactivity with the user
- htmx addresses each of these shortcomings, increasing the expressiveness of HTML as a hypermedia
- The `hx-get`, `hx-post`, etc. attributes can be used to issue requests with any element in the dom
- The `hx-swap` attribute can be used to control exactly how HTML responses to htmx requests should be swapped into the DOM
- The `hx-trigger` attribute can be used to control the event that triggers a request
- Event filters can be used in `hx-trigger` to narrow down the exact situation that you want to issue a request for
- htmx offers three mechanisms for including additional input information with requests:
  - Enclosing elements within a `form` tag
  - Using the `hx-include` attribute to select inputs to include in the request
  - `hx-vals` for embedding values directly via JSON or, dynamically, resolving values via JavaScript
- htmx also provides integration with the browser history and back button, using the `hx-push-url` attribute