Hypermedia In Action

1. Extending HTML As Hypermedia

1.1. The Shortcomings of 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. This application was built using nothing but forms and anchor tags, exchanging hypermedia (HTML) with the server via HTTP.

This is a fine, if simple, application, but it isn't what many people expect from the web today. It suffers from a few problems typical of early Web 1.0 applications:

- There is a noticeable refresh when you move between pages of the application, of when you create, update or delete a contact.
- All the updates are done with the `POST` HTTP action, despite the presence of more logical actions like `PUT` and `DELETE`

The first point, in particular, is noticeable in Web 1.0 applications and is what is responsible for giving them the reputation for being "clunky" when compared with JavaScript-based Single Page Applications. SPAs eliminates this clunkiness by updating the web page directly. It does this by mutating the Document Object Model (DOM), the JavaScript API to the underlying HTML page. There are many different implementations of this approach, but the most common today is to tie the DOM to a JavaScript model and *reactively* update the DOM when the JavaScript model is updated.

As we discussed in Chapter 1, communication with the server in this situation 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 applications. We think that this is a bit close minded, but we do understand the genuine concerns developers have regarding the user experience of these older Web 1.0 applications.

It is worth noting that there is nothing *inherent* in the concept of hypermedia that dictates there must be a clunky user experience.

1.1.1. A Close Look At A Link

To see how the hypermedia approach to building web applications might be generalized to address these UX concers, let's reconsider the anchor tag from Chapter 1, which creates a web link to the Manning website:

Listing 3. 1. A Simple Hyperlink, Again

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

Let's consider what this link tells a browser to do, this time in excruciating detail:

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

These are the four aspects of a simple hypermedia link, the last three being the mechanic that distinguishes a link from "normal" text.

So let's spend a moment and think about generalizing this hypermedia mechanic within HTML. There is no rule saying that hypermedia can *only* work this way, after all! So, what if it were the case that any element could issue a HTTP request. For example, shouldn't `button` elements be able to issue requests? Why should only anchor tags and forms be able to do so, as in plain HTML?

And what's so special about clicking (in the case of anchors) or submitting (in the case of forms)? Those are just one of many events that are fired by the DOM! Why shouldn't other events be able to trigger requests as well?

And why does HTML only give us access to the `GET` and `POST` actions of HTTP? HTTP *stands* for HyperText Transfer Protocol, and the format it was designed for, HTML, only supports two of the five developer-facing request types! You *have* to use JavaScript to get at the other three, like `DELETE`. That's ridiculous, that should be fixed!

Finally, and this is perhaps the most important conceptual extension of HTML: why should it be necessary to replace the *entire* screen when an HTTP request is made by an element in HTML? Normal HTML actually has a mechanism for avoiding this, iframes, but they are extremely limited and have largely fallen out of favor except in a few specialized applications. What if HTML made it possible to replace only *part* of the DOM with the response? This would make HTML function more like an SPA, where only part of the DOM was updated, giving a much smoother overall experience.

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

1.2. Extending HTML as a Hypermedia with htmx

It turns out that there are JavaScript libraries that, in fact, extend HTML in exactly this manner. It is 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! There is no rule saying it has to be used to write SPAs and, in fact, it is the perfect tool for addressing some of 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 approach, but it is perhaps the purest in the pursuit of extending HTML as a hypermedia. It focuses on the issues discussed above and attempts to incrementally address each one, without introducing a significant amount of additional conceptual infrastructure.

This is not without tradeoffs: by staying so close to HTML, htmx does not give you a lot of infrastructure that many developers might feel should be there "by default". A good example is the notion of modals: many web applications today make heavy use of modal dialogs that introduce client-side state (namely, a modal window, or "mode") in a manner that is, frankly, inconsistent with the stateless philosophy of the web. That's not to say you can't use modals with htmx, and we will look at how you can do so later, but rather to say that htmx, just like HTML, won't make your life easy in this case. htmx *can* be used to effectively implement a 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.

1.2.1. Using htmx

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

Listing 3. 2. Installing htmx

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

</head>
```

Here we are using the popular unpkg Content Delivery Network (CDN) to install version `1.7.0` of the library. We are also using 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.

That's all it takes to install htmx!

Of course, you may not want to use a CDN, in which case you can download htmx to your local system and adjust this script tag. Or, you may have a more elaborate 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. Now, unlike the vast majority of JavaScript libraries, htmx does not require you, the user, to actually write 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 attributes to specify which URL should be the target of an htmx-driven HTTP request.

1.3. Triggering HTTP Requests

The core of htmx consists of five attributes that can be used to issue the five major 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 something else) 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. 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">
  Get The Contacts
</button>
```

The htmx library will see this attribute and hook up some JavaScript logic to issue an HTTP `GET` AJAX request to the `/contacts` path when the user clicks on this button.

Here we get to the most important thing to understand about htmx: it expects the response to this AJAX request *to be HTML*, not JSON! htmx is an extension of HTML and, just as the response to an anchor tag click or form submission is typically HTML, htmx expects the server to respond with a hypermedia, namely HTML.

One important difference between the HTTP responses to normal anchor and form driven requests and htmx driven requests is that, in the case of htmx, responses are often only *partial* bits of HTML. Since we are not replacing the whole document it is not necessary to transfer the entire HTML document from the server to the browser. This can be used to save bandwidth as well as resource loading time, since less overall content is transferred and since it isn't necessary to reprocess a `head` tag with style sheets, script tags, and so forth.

A simple response to the above 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 list with some clickable elements in it, a real response would of course likely contain more information. But this simple response demonstrates that htmx is staying within the hypermedia paradigm: once again 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.

1.4. Targeting Other Elements

Now, given that htmx has issued a request and gotten back some HTML as a response, what should it do with it? Well, the default behavior is to simply put the returned content into the element that triggered the request. That's obviously *not* a good thing in this situation: we would end up with a list of contacts awkwardly embedded within a button element on the page!

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

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">

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

</div>
```

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.

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

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 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 and element or an entire element respectively. The next four values are taken from the `Element.insertAdjacentHTML()` DOM API. The last two are specific to htmx, but are fairly obvious to understand. Again, htmx tries to stay as close as possible to the web standards to keep your conceptual load to a minimum.

So, lets consider if, rather than replacing the `innerHTML` content of the main div above, we wished to replace the entire div with the 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">
    Get The Contacts
  </button>

</div>
```

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>
```

The target div has been entirely removed from the DOM, and the list has taken its place.

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

1.5. 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! And, yes, by default when an `hx-get` or another request-driving annotation is placed on a button, the request will be issued 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 and 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 just take 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">
    Get The Contacts
  </button>

</div>
```

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">
    Get The Contacts
  </button>

</div>
```

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']">
    Get The Contacts
  </button>

</div>
```

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 itself*, so you can easily filter on properties of a given event. You can use any expression you like for a filter, however: a global JavaScript function 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. This is obviously not what we want! 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.

Here we want to listen to the `keyup` events on the entire page, or, equivalently, on the `body` element. The `from:` modifier, as with many other attributes and modifiers in htmx, uses a CSS selector to select the element to listen on, and can be used 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">
    Get The Contacts
  </button>

</div>
```

Now our button is listening for the `keyup` event on the body of the page, and should issue a request

no only when it is clicked on, but 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 defaults 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 check it out! With this final attribute, `hx-trigger`, we have addressed *all* of the shortcomings of HTMl that we considered at the start of this chapter. That's a grand total of eight, count 'em, *eight* attributes that all fall within the same conceptual model as normal HTML and, by extending HTML as a hypermedia, open up world of new user interface possibilities!

1.6. 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 that other element in traditional hypermedia-based applications, forms, which are used to pass additional information beyond just the URL up to the server in a request. This information is typically collected via various types of input tags in HTML. htmx allows you include these additional request parameters in a natural way that mirrors how HTML itself works.

1.6.1. Enclosing Forms

The simplest way to pass additional parameters up with a request in htmx is to enclose it, as well as the inputs that collect those parameters, 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>
      <label for="search">Search Contacts:</label>
      <input id="search" name="q" type="search" placeholder="Search Contacts">
      <button hx-post="/contacts" hx-target="#main">
        Search The Contacts
      </button>
  </form>

</div>
```

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 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 inclosing form for `GET` requests. This is to avoid serializing forms in situations that the data is not needed and to keep URLs clean when dealing with history entries, discussed next.

1.6.2. Including inputs

Another mechanism for including value in requests is to use the `hx-include` attribute to select input values that you wish to include in a request. 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">
    Search The Contacts
  </button>

</div>
```

Note that the `hx-include` attribute takes a CSS selector 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.

1.6.3. 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"}'>
  Get The Contacts In Montana
</button>
```

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.

Using these mechanisms you can include values in your hypermedia requests with htmx in a manner that is very familiar and in keeping with the original HTML model.

1.7. 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">
  Get The Contacts
</button>
```

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.

1.7.1. Distinguishing Between htmx & Regular HTTP Requests

Now, if you are following closely, you may have noticed a problem here: we have updated the the URL in the browsers nagivation bar to now have the path `/contacts` in it, but htmx typically gets back *partial* bits of HTML. Recall that the response in this case might look only like this:

Listing 3. 18. 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>
```

But what if someone hits the refresh button on this page? A new `GET` request will be issued to the `/contacts` path and, if we don't do anything differently, we would just render this little bit of HTML and present the user with a very ugly list of contacts without any surrounding HTML!

Fortunately htmx helps you deal with this situation by including an HTTP Request Header, `HX-Request`. You can look for this header and, if it is present, you can render the partial bit of HTML, and if it is not, you can render the entire HTML page. This allows you to easily handle both htmx and regular HTTP requests to this URL cleanly.

Depending on the server side templating library you use, dealing with this can often be as easy as deciding whether or not to include a layout when rendering a given bit of content.

1.8. Summary

In this chapter we considered some shortcomings of HTML as a hypermedia and looked at how htmx can help to rectify them:

- HTML doesn't give you access to non-`GET` or `POST` requests
- HTML requires that you update the entire script
- HTML only offers limited interactivity
- htmx addresses all of these shortcomings, increasing the expressiveness of HTML as a hypermedia

In the next chapter we will take the tools we have learned about here and, using them, improve the simple Web 1.0 application we looked at in Chapter 2. I think you will be surprised at just how much we can improve the functionality and usability of the application with our more powerful

hypermedia!