Hypermedia In Action

1. Advanced Hypermedia Patterns

This chapter covers:

- Adding the "Active Search" pattern to our application
- Adding the "Lazy Load" pattern to our application
- Implementing inline deletion of contacts from the list view
- Implementing a bulk delete of contacts

1.1. Active Search

In this chapter we will add some more advanced features to our contacts application, all while staying within the hypermedia model. (We will do some client-side scripting in our application later on, but, even when we do add scripting based features, we will keep the network communication model firmly within the hypermedia architecture!)

The first advanced feature we will create is known as the "Active Search" pattern. Active Search is a feature when, as a user types text into a search box, the results of that search are dynamically updated. This pattern was made popular when Google adopted it for search results, and many applications now implement it.

As you might suspect, we are going to use some of the same techniques we used for dynamically updating emails in the previous chapter, since we are once again going to want to issue requests on the `keyup` event.

Let's look at the current search field in our application once again:

Listing 5. 1. Our Search Form

```
<form action="/contacts" method="get" class="tool-bar">
    <label for="search">Search Term</label>
    <input id="search" type="search" name="q" value="{{ request.args.get('q') or ''
}}"/>
    <input type="submit" value="Search"/>
</form>
```

You will recall that we have some server side code that looks for the `q` parameter and, if it is present, searches the contacts for that term.

As it stands right now, the user must hit enter when the search input is focused, or click the "Search" button. Both of these events will trigger a `submit` event on the form, causing it to issue an HTTP `GET` and re-rendering the whole page. Recall that currently, thanks to `hx-boost` the form will still use an AJAX request for this `GET`, but we don't get the nice search-as-you-type behavior we want.

To add active search behavior, we will need to add a few htmx attributes to the search input. We will leave the form as is, so that, in case a user does not have JavaScript enabled, search continues to work. (As a reminder, this is called "progressive enhancement" and this pattern is progressive.) We want to issue an HTTP `GET` request to the same URL that the form does when it is submitted. And we want to do so when a key up occurs, but only after a small delay. We can take the `hx-trigger` attribute directly from our email validation example!

Listing 5. 2. Adding Active Search Behavior

```
<form action="/contacts" method="get" class="tool-bar">
    <label for="search">Search Term</label>
    <input id="search" type="search" name="q" value="{{ request.args.get('q') or ''
}}" ❶
           hx-get="/contacts" ❷
           hx-trigger="search, keyup delay:200ms changed"/> ❸
    <input type="submit" value="Search"/>
</form>
```

❶ We keep everything the same on the input, so it functions the way it always has if JavaScript isn't enabled

❷ We issue a `GET` to the same URL as the form

❸ We use a similar `hx-trigger` specification as we did for the email input validation example

The small change that we made to the `hx-trigger` attribute is we switched out the `change` event for the `search` event. The `search` event is triggered when someone clears the search or hits the enter key. It is a non-standard event, but doesn't hurt to include here. The main functionality of the feature is provided by the second triggering event, the `keyup` which, as with the email example, is delayed to debounce the input requests and avoid

issuing too many requests.

What we have is pretty close to what we want, but recall that the default target for an element is itself. As things currently stand, an HTTP `GET` request will be issued to the `/contacts` path, which will, as of now, return an entire HTML document of search results! This whole document will then be inserted into the inner HTML of an input! Well, that's pretty meaningless, and the browser will, sensibly, just ignore htmx's request to do this. So, at this point, when a user types anything into our input, a request will be issued, but it will appear to the user as if nothing has happened.

OK, so to fix this issue, what do we want to target with the update instead? Ideally we'd like to just target the actual results: there is no reason to update the header or search input, and that could cause an annoying flash as focus jumps around.

Fortunately the `hx-target` attribute allows us to do exactly that! Lets use it to target the results body, the `tbody` element in the table of contacts:

Listing 5. 3. Adding Active Search Behavior

```
<form action="/contacts" method="get" class="tool-bar">
    <label for="search">Search Term</label>
    <input id="search" type="search" name="q" value="{{ request.args.get('q') or ''
}}"
           hx-get="/contacts"
           hx-trigger="change, keyup delay:200ms changed"
           hx-target="tbody"/> ❶
    <input type="submit" value="Search"/>
</form>
```

❶ Target the `tbody` tag on the page

Because there is only one `tbody` on the page, we can use the CSS selector `tbody` and htmx will target the first element matching that selector.

Now if you try typing something into the search box, you'll get some action. A request is made and the results are inserted into the document within the tbody. Unfortunately, the results are… the entire document still! So you get all the other stuff, the search box, the application header, etc. and a somewhat humorous double-render.

Now, we could use the same trick we reached for in the "Click To Load" and "Infinite

Scroll" features: `hx-select`. Recall that `hx-select` allows us to pick out the part of the response we are interested in using CSS selectors. So we could add this to our input:

Listing 5. 4. Using `hx-select` for Active Search

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

❶ Adding an `hx-select` that picks out the table rows in the `tbody` of the response

1.1.1. Server Side Tricks With htmx

This works fine, but we are not going to use this approach. Here we are letting the server create a full HTML document in response and then, on the client side, filtering it down. This is easy and might be necessary if we don't control the server side or can't easily modify responses. But here we can modify our server responses so, instead of using this client-side approach, we are going to use this as an opportunity to explore returning different bits of HTML based on the context information that htmx provides with requests.

Let's take a look again at the server side code for our search logic:

Listing 5. 5. Server Side Search

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    if search:
        contacts_set = Contact.search(search) ❶
    else:
        contacts_set = Contact.all()
    return render_template("index.html", contacts=contacts_set) ❷
```

❶ This is where the search logic happens

❷ We simply rerender the `index.html` template every time, no matter what

What we want to do on the server side is *conditionally* render only the table rows when we are serving an "Active Search" request. Remember, though, we *also* need to handle "regular" search requests submitted by the form, in case JavaScript is disabled, or the user

clicks the "Search" button. In these cases we want the current logic, where we render the entire `index.html` template, to execute.

So we need some way to determine exactly *who* made the request to the `/contact` URL to know what to render. It turns out that htmx helps us out here by including a number of HTTP *Request Headers* when it makes requests. Request Headers are name/value pairs of metadata associated with the request and are a standard, if underutilized, feature of HTTP.

Here are the headers that htmx gives us to work with:

Header	**Description**
HX-Boosted	This will be the string "true" if the request is made via an element using hx-boost
HX-Current-URL	This will be the current URL of the browser
HX-History-Restore-Request	This will be the string "true" if the request is for history restoration after a miss in the local history cache
HX-Prompt	This will contain the user response to an hx-prompt
HX-Request	This value is always "true" for htmx-based requests
HX-Target	This value will be the id of the target element if it exists
HX-Trigger-Name	This value will be the name of the triggered element if it exists
HX-Trigger	This value will be the id of the triggered element if it exists

Looking through this list of headers, the last one stands out: we have an id, `search` on our search input. So the value of the `HX-Trigger` header should be set to `search` when the request is coming from the search input. Perfect!

Let’s add some conditional logic to our controller:

Listing 5. 6. Updating Our Server Side Search

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    if search:
        contacts_set = Contact.search(search)
        if request.headers.get('HX-Trigger') == 'search': ❶
          ??? ❷
    else:
        contacts_set = Contact.all()
    return render_template("index.html", contacts=contacts_set) ❷
```

❶ If the request header `HX-Trigger` is equal to "search", we want to do something different

❷ But what is that something?

OK, we have the conditional logic in place in our controller, but what do we want to do here? Well, we want to do something akin to what we were achieving using `hx-select` previously: we only want to render the *rows* of the table within the table body!

How can we achieve that?

1.1.2. Factoring Your Templates

Here we come to a common pattern in htmx: we want to *factor* our server side templates. This means that we want to break them up a bit so they can be called from multiple contexts. In this situation, we want to break the rows of the results table out to a separate template. We will call this new template `rows.html` and we will include it from the main `index.html` template, as well as render it directly in the controller when we want to respond with only the rows to Active Search requests.

Recall what the table in our `index.html` file currently looks like:

Listing 5. 7. The Contacts Table

```
<table>
    <thead>
    <tr>
        <th>First</th> <th>Last</th> <th>Phone</th> <th>Email</th> <th></th>
    </tr>
    </thead>
    <tbody>
    {% for contact in contacts %} ❷
        <tr>
            <td>{{ contact.first }}</td>
            <td>{{ contact.last }}</td>
            <td>{{ contact.phone }}</td>
            <td>{{ contact.email }}</td>
            <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
                <a href="/contacts/{{ contact.id }}">View</a></td>
        </tr>
    {% endfor %}
    </tbody>
</table>
```

What we want to do is to move that for loop and the rows it creates out so a separate file, and save that as `row.html`:

Listing 5. 8. Our New `rows.html` file

```
    {% for contact in contacts %} ❷
        <tr>
            <td>{{ contact.first }}</td>
            <td>{{ contact.last }}</td>
            <td>{{ contact.phone }}</td>
            <td>{{ contact.email }}</td>
            <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
                <a href="/contacts/{{ contact.id }}">View</a></td>
        </tr>
    {% endfor %}
```

We can then include this new file in our table in `index.html` by using the Jinja2 `include` directive:

Listing 5. 9. Including The New File

```
    <table>
        <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Phone</th>
            <th>Email</th>
            <th></th>
        </tr>
        </thead>
        <tbody>
        {% include 'rows.html' %} ❶
        </tbody>
    </table>
```

❶ This directive includes the `rows.html` file, inserting the content from that template into the `index.html` template

So far, so good. The application still works and if we navigate to the `/contacts` page, everything is still rendering properly. But we need to go back and fix up our controller now to take advantage of this new file when we are doing an Active Search. Luckily, the update is simple: we just need to call the `render_template` function with this new file:

Listing 5. 10. Updating Our Server Side Search

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    if search:
        contacts_set = Contact.search(search)
        if request.headers.get('HX-Trigger') == 'search':
          return render_template("rows.html", contacts=contacts_set) ❶
    else:
        contacts_set = Contact.all()
    return render_template("index.html", contacts=contacts_set)
```

❶ Render the new template in the case of an active search

Now, when an Active Search request is made, rather than getting an entire HTML document back, we only get a partial bit of HTML, the table rows for the contacts that

match the search. These rows are then inserted into the `tbody` on the index page, without any need for an `hx-select` or any other client side processing.

And the old form-based search still works as well, thanks to the fact that we conditionally render the rows only when the `search` input issues the HTTP request.

1.1.3. Updating History

You may have noticed one shortcoming of our Active Search when compared with submitting the form: the form puts the query into the navigation bar as a URL parameter. So if you search for "joe" in the search box, you will end up with a url that looks like this:

`https://example.com/contacts?q=joe`

This features makes it such that you can copy the URL and send it to someone else, and they can simply click on the link to repeat the exact same search. As it stands right now, during Active Search, we do not update the URL.

Let's fix that by adding the `hx-push-url` attribute:

Listing 5. 11. Updating The URL During Active Search

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

❶ By adding the `hx-push-url` attribute with the value `true`, htmx will update the URL when it makes a request

That's all it takes and now, as Active Search requests are sent, the URL in the browser is updated to have the query in it, just like when the form is submitted.

Now, you might not *want* this behavior. You might feel it would be confusing to users to see the navigation bar updated and have history entries for every Active Search made, for example. Which is fine! You can simply omit the `hx-push-url` attribute and it will go back ot the behavior you want. htmx tries to be flexible enough that you can achieve the UX you want, while staying largely within the declarative HTML model.

1.1.4. Adding A Request Indicator

A final touch for our Active Search pattern is to add a request indicator to let the user know that a search is in progress. As it stands the user has to know that the active search functionality is doing a request implicitly and, if the search takes a bit, may end up thinking that the feature isn't working. By adding a request indicator we let the user know that the hypermedia application is busy and they can wait (hopefully not too long!) for the request to complete.

htmx provides support for request indicators via the `hx-indicator` attribute. This attribute takes, you guessed it, a CSS selector that points to the indicator for a given element. The indicator can be anything, but it is typically some sort of animated image, such as a gif or svg file, that spins or otherwise communicates visually that "something is happening".

Let's add a spinner next to our input:

Listing 5. 12. Updating The URL During Active Search

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or '' }}"
       hx-get="/contacts"
       hx-trigger="change, keyup delay:200ms changed"
       hx-target="tbody"
       hx-push-url="true"
       hx-indicator="#spinner"/> ❶
<img id="spinner" class="htmx-indicator" src="/static/img/spinning-circles.svg"
alt="Request In Flight..."/> ❷
```

❶ The `hx-indicator` attribute points to the indicator image after the input

❷ The indicator is a spinning circle svg file, and has the `htmx-indicator` class on it

We have added the spinner right after the input. This visually co-locates the request indicator with the element making the request, and makes it easy for a user to see that something is in fact happening.

Note that the indicator `img` tag has the `htmx-indicator` class on it. This is a CSS class automatically injected by htmx that defaults the element to an `opacity` of 0. When an htmx request is triggered that points to this indicator, another class, `htmx-request` is added to the indicator which transitions its opacity to 1. So you can use just about anything

as an indicator and it will be hidden by default, and will be shown when a request is in flight. This is all done via standard CSS classes, allowing you to control the transitions and even the mechanism by which the indicator is show (e.g. you might use `display` rather than `opacity`). htmx is flexible in this regard.

Request indicators are an important UX aspect of any distributed application. It is unfortunate that browsers have de-emphasized their native request indicators over time, and it is doubly unfortunate that request indicators are not part of the JavaScript ajax APIs.

Be sure not to neglect this important aspect of your application! Even though requests might seem instant when you are working on your application locally, in the real world