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.

Use Request Indicators!

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

So there we go: we now have a pretty darned sophisticated user experience built out when compared with plain HTML, but we've built it all as a hypermedia-driven feature, no JSON or JavaScript to be seen! This particular implementation also has the benefit of being a progressive enhancement, so this aspect of our application will continue to work for clients that don't have JavaScript enabled. Pretty slick!

1.2. Lazy Loading

With active search behind us, let's move on to a very different sort of problem, that of lazy loading. Lazy loading is when the loading of something is deferred until later, when needed. This is commonly used as a performance enhancement: you avoid the processing resources necessary to produce some data until that data is actually needed.

Let's add a count of the total number of contacts below the bottom of our contacts table. This will give us a potentially expensive operation that we can use to demonstrate how easy it is to add lazy loading to our application using htmx.

First let's update our server code in the `/contacts` request handler to get a count of the total number of contacts. We will pass that count through to the template to render some new HTML.

Listing 5. 13. Adding A Count To The UI

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    page = int(request.args.get("page", 1))
    count = Contact.count() ❶
    if search:
        contacts_set = Contact.search(search)
        if request.headers.get('HX-Trigger') == 'search':
            return render_template("rows.html", contacts=contacts_set, page=page,
count=count)
    else:
        contacts_set = Contact.all(page)
    return render_template("index.html", contacts=contacts_set, page=page,
count=count) ❷
```

❶ Get the total count of contacts from the Contact model

❷ Pass the count out to the `index.html` template to use when rendering

Now, as with the rest of the application, we are not going to focus on the internals of `Contact.count()`, we are just going to take it for granted that:

- It returns the total count of contacts in the contact database
- It may potentially be slow

Next lets add some HTML to our `index.html` that takes advantage of this new bit of data, showing a message next to the "Add Contact" link with the total count of users. Here is what our HTML looks like:

Listing 5. 14. Adding A Contact Count Element To The Application

```
<p>
    <a href="/contacts/new">Add Contact</a> <span>({{ count }} total
Contacts)</span>❶
</p>
```

❶ A simple span with some text showing the total number of contacts.

Well that was easy, wasn't it? Now our users will see the total number of contacts next to the link to add new contacts, to give them a sense of how large the contact database is. This

sort of rapid development is one of the joys of developing web applications the old way.

Here is what the feature looks like in our application:

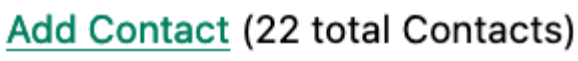

Figure 5. 1. Total Contact Count Display

Beautiful.

Of course, as you probably suspected, all it not perfect. Unfortunately, upon shipping this feature to production, we start getting some complaints from the users that the application "feels slow". So, like all good developers faced with a performance issues, rather than guessing what the issue might be, we try to get a performance profile of the application to see what exactly is causing the problem.

It turns out, surprisingly, that the problem is that innocent looking `Contacts.count()` call, which is taking up to a second and a half to complete. Unfortunately, for reasons beyond the scope of this book, it is not possible to improve that load time, nor it is also not possible to cache the result. That leaves us with two choices:

- Remove the feature
- Come up with some other way to mitigate the performance issue

After talking with the project manager, it is clear that removing the feature isn't an acceptable solution, so we will need to take another approach to mitigating the performance issue. And the approach we will use is lazy loading, where we defer loading the count "until later". Let's look at exactly how we can accomplish this using htmx.

1.2.1. Pulling The Expensive Code Out

The first step is to pull the expensive code, that is, the call to `Contacts.count()` out of request handerl for the `/contacts` end point.

Let's move it to a new end point at `/contacts/count` instead. We won't need to render a template with this new end point. It's only job is going to be to render that small bit of text that is in the span, "(22 total Contacts)"

Here is what the new code will look like:

Listing 5. 15. Pulling The Expensive Code Out

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    page = int(request.args.get("page", 1)) ❶
    if search:
        contacts_set = Contact.search(search)
        if request.headers.get('HX-Trigger') == 'search':
            return render_template("rows.html", contacts=contacts_set, page=page)
    else:
        contacts_set = Contact.all(page)
    return render_template("index.html", contacts=contacts_set, page=page) ❷

@app.route("/contacts/count")
def contacts_count():
    count = Contact.count() ❸
    return "(" + str(count) + " total Contacts)" ❹
```

❶ We no longer call `Contacts.count()` in this handler

❷ `count` is no longer passed out to the template to render in the `/contacts` handler

❸ We create a new handler at the `/contacts/count` path that does the expensive calculation

❹ Return the string with the total number of contacts in it

Great! Now we have an end point that will produce that expensive-to-create text for us. The next step is to hook up the span we have created to actually retrieve this text. As we said earlier, the default behavior of htmx is to place any content it recieves for a given request into the `innerHTML` of an element, which is exactly what we want here: we want to retrieve this text and put it into the `span`. We can use the `hx-get` attribute pointing to this new path to do exactly that.

However, recall that the default *event* that will trigger a request for a `span` element in htmx is the `click` event. That's not what we want. Instead, we want this to trigger passively, when the page loads. To do this, we can use the `hx-trigger` attribute with the value `load`. This will cause htmx to issue the request when the element is loaded into the page.

Here is our updated template code:

Listing 5. 16. Adding A Contact Count Element To The Application

```
<p>
    <a href="/contacts/new">Add Contact</a> <span hx-get="/contacts/count" hx-
trigger="load"></span>❶
</p>
```

❶ Issue a `GET` to `/contacts/count` when the `load` event occurs

Note that the `span` starts empty: we have removed the content from it, and we are allowing the request to `/contacts/count` to populate it instead.

And, lo and behold, our `/contacts` page is fast again! When you navigate to the page it feels very snappily and profiling shows that yes, indeed, the page is loading much more quickly. Why is that? We'll we've deferred the expensive calculation to a secondary request, allowing the initial request to finish loading much more quickly.

You might say "OK, great, but it's still taking a second or two to get the total count on the page." That's true, but often the user may not be particularly interested in the total count. They may just want to come to the page and search for an existing user, or perhaps they may want to edit or add a user. The total count is often just a "nice to have" bit of information in these cases. By deferring the calculation of the count in this manner we let users get on with their use of the application while we perform the expensive calculation.

Yes, the total time to get all the information on the screen takes just as long. (It actually might be a bit longer since we now have two requests that need to get all the information.) But the *percieved performance* for the end user will be much better: they can do what they want nearly immediately, even if some information isn't available instantaneously. This is a great tool to have in your toolbelt when optimizing your web application performance!

1.2.2. Adding An Indicator

Unfortunately there is one somewhat disconcerting aspect to our current implementation: the count is lazily loaded, but there is no way for a user to know that this computation is being done. As it stands, the count just sort of bursts onto the scene whenever the request to `/contacts/count` completes.

That's not ideal. What we want is an indicator, like we added to our active search example.

And, in fact, we can simply reuse the same spinner image here!

Now, in this case, we have a one-time request and nce the request is over, we are not going to need the spinner anymore. So it doesn't make sense to use the exact approach we did with the active search example, placing a spinner beside the span and using the `hx-indicator` attribute to point to it.

Instead, we can put the spinner *inside* the content of the span. When the request completes the content in the response will be placed inside the span, replacing the spinner with the computed contact count. It turns out that htmx allows you to place indicators with the `htmx-indicator` class on them inside of elements that are htmx-powered and, in the absence of an `hx-indicator` attribute, these internal indicators will be shown when a request is in flight.

So let's add that spinner from the active search example as the initial content in our span:

Listing 5. 17. Adding An Indicator To Our Lazily Loaded Content

```
<span hx-get="/contacts/count" hx-trigger="load">
  <img id="spinner" class="htmx-indicator" src="/static/img/spinning-circles.svg"/>❶
</span>
```

❶ Yep, that's it

Great! Now when the user loads the page, rather than having the total contact count sprung on them like a surprise, there is a nice spinner indicating that something is coming. Much better!

Note that all we had to do was copy and paste our indicator from the active search example into the `span`! This is a great demonstration of how htmx provides flexible, composable features and building blocks to work with: implementing a new feature is often just a copy-and-paste, with maybe a tweak or two, and you are done.

1.2.3. But That's Not Lazy!

You might say "OK, but that's not really lazy. We are still loading the count immediately when the page is loaded, we are just doing it in a second request. You aren't really waiting until the value is actually needed."

Fine. Let's make it *lazy* lazy: we'll only issue the request when the span scrolls into view.

To do that, just recall how we set up the infinite scroll example: we used the `revealed` event for our trigger. That's all we want here, right? When the element is revealed we issue the request.

Listing 5. 18. Making It Lazy Lazy

```
<span hx-get="/contacts/count" hx-trigger="revealed"> ❶
  <img id="spinner" class="htmx-indicator" src="/static/img/spinning-circles.svg"/>
</span>
```

❶ Yep, that's it

Now we have a truly lazy implementation, deferring the expensive computation until we are absolutely sure we need it. A pretty cool trick, and, again, a simple one-attribute change demonstrates the flexibility of both htmx the hypermedia approach.

1.3. Inline Delete

We now have some pretty slick UX patterns in our application, but let's not rest on our laurels. For our next hypermedia trick, we are going to implement "inline delete", where a contact can be deleted directly from the list view of all contacts, rather than requiring the user to drill in to the edit view of particular contact to access the "Delete Contact" button.

We already have "Edit" and "View" links for each row, in the `rows.html` template:

Listing 5. 19. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
</td>
```

We want to add a "Delete" link as well. And we want that link to act an awful lot like the "Delete Contact" from `edit.html`, don't we? We'd like to issue an HTTP `DELETE` to the URL for the given contact, we want a confirmation dialog to ensure the user doesn't accidentally delete a contact. Here is the "Delete Contact" html:

Listing 5. 20. The Existing Row Actions

```
<button hx-delete="/contacts/{{ contact.id }}"
        hx-push-url="true"
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="body">
    Delete Contact
</button>
```

Is this going to be another copy-and-paste job with a bit of tweaking? It sure is!

One thing to note is that, in the case of the "Delete Contact" button, we want to rerender the whole screen and update the URL, since we are going to be returning from the edit view for the contact to the list view of all contacts. In the case of this link, however, we are already on the list of contacts, so there is no need to update the URL, and we can omit the `hx-push-url` attribute.

Here is our updated code:

Listing 5. 21. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
    <a href="#" hx-delete="/contacts/{{ contact.id }}"
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="body">Delete</a> ❶
</td>
```

❶ Almost a straight copy of the "Delete Contact" button

As you can see, we have added a new anchor tag and given it a blank target (the `#` value in its `href` attribute) to retain the correct mouse-over styling behavior of the link. We've also copied the `hx-delete`, `hx-confirm` and `hx-target` attributes from the "Delete Contact" button, but omitted the `hx-push-url` attributes since we don't want to update the URL of the browser.

And… that's it! We now have inline delete working, even with a confirmation dialog!

1. A Style Sidebar

One thing is really starting to bother me about our application: we now have quite a few actions stacking up in our contacts table, and it is starting to look very distracting:

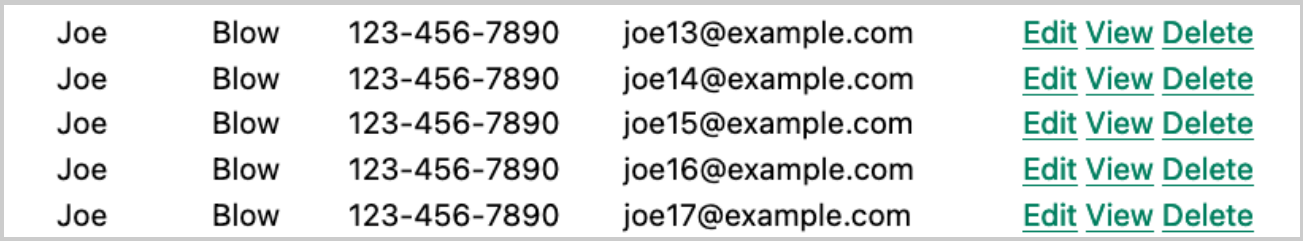

Figure 5. 2. That's a Lot of Actions

It would be nice if we didn't show the actions all in a row, and it would be nice if we only showed the actions when the user indicated interest in a given row. We will return to this problem after we look at the relationship between scripting and a Hypermedia Driven Application in a later chapter.

For now, let's just tolerate this less-than-ideal user interface, knowing that we will return to it later.

1.3.1. Getting Fancy

We can get even fancier here, however. What if, rather than re-rendering the whole page, we just removed the row for the contact? The user is looking at the row anyway, so is there really a need to re-render the whole page?

To do this, we'll need to do a couple of things:

- We'll need to update this link to target the row that it is in
- We'll need to change the swap to `outerHTML`, since we want to replace (really, remove) the entire row
- We'll need to update the server side to render empty content when the `DELETE` is issued from a row rather than from the "Delete Contact" button on the contact edit page

First things first, update the target of our "Delete" link to be the row that the link is in, rather than the entire body. We can once again take advantage of the relative positional `closest` feature to target the closest `tr`, like we did in our "Click To Load" and "Infinite Scroll" features:

Listing 5. 22. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
    <a href="#" hx-delete="/contacts/{{ contact.id }}"
        hx-swap="outerHTML"
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="closest tr">Delete</a> ❶
</td>
```

❶ Updated to target the closest enclosing `tr` (table row) of the link

Now we need to update the server side as well. We want to keep the "Delete Contact" button working as well, and in that case the current logic is correct. So we'll need some way to differentiate between `DELETE` requests that are triggered by the button and `DELETE` requests that come from this anchor.

The cleanest way to do this is to add an `id` attribute to the "Delete Contact" button, so that we can inspect the `HX-Trigger` HTTP Request header to determine if the delete button was the cause of the request. This is a simple change to the existing HTML:

Listing 5. 23. Adding an `id` to the "Delete Contact" button

```
    <button id="delete-btn" ❶
            hx-delete="/contacts/{{ contact.id }}"
            hx-push-url="true"
            hx-confirm="Are you sure you want to delete this contact?"
            hx-target="body">
        Delete Contact
    </button>
```

❶ An `id` attribute has been added to the button

With this in place, we now have a mechanism for differentiating between the delete button in the `edit.html` template and the delete links in the `rows.html` template. We can write code very similar to what we did for the active search pattern, using a conditional on the `HX-Trigger` header to determine what we want to do. If that header has the value `delete-btn`, then we know the request came from the button on the edit page, and we can do what we are currently doing: delete the contact and redirect to `/contacts` page.

If it does not have that value, then we can simple delete the contact and return an empty string. This empty string will replace the target, in this case the row for the given contact, thereby removing the row from the UI.

Let's make that change to our server side code:

Listing 5. 24. Updating Our Server Code To Handle Two Different Delete Patterns

```
@app.route("/contacts/<contact_id>", methods=["DELETE"])
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete()
    if request.headers.get('HX-Trigger') == 'delete-brn': ❶
        flash("Deleted Contact!")
        return redirect("/contacts", 303)
    else:
        return "" ❷
```

❶ If the delete button on the edit page submitted this request, then continue to do the logic we had previous

❷ If not, simply return an empty string, which will delete the row

Believe it or not, we are now done: when a user clicks "Delete" on a contact row and confirms the delete, the row will disappear from the UI. Poof! Once again, we have a situation where just changing a few lines of simple code gives us a dramatically different behavior. Hypermedia is very powerful!

1.3.2. Getting Fancy Fancy With The htmx Swapping Model

This is pretty cool, but there is another improvement we can make if we take some time to understand the htmx content swapping model: it sure would be sexy if, rather than just instantly deleting the row, we faded it out before we removed it. That easement makes it more obvious that the row is being removed, giving the user some nice visual feedback on the modification.

It turns out we can do this pretty easily with htmx, but to do so we'll need to dig in to exactly how htmx swaps content. You might think that htmx simply puts the new content into the DOM, but that's not how it works. Instead, content goes through a series of steps as it is added to the DOM:

- When content is recieved and about to be swapped into the DOM, the `htmx-swapping` CSS class is added to the target element
- A small delay then occurs (we will discuss why in a moment)
- Next, the `htmx-swapping` class is removed from the target and the `htmx-settling` class is added
- The new content is swapped into the DOM
- Another small delay occurs
- Finally, the `htmx-settling` class is removed from the target

There is more to the swap mechanic, settling for example is a more advanced topic that we will discuss in a later chapter, but for now this is all you need to know about the swapping mechanism. Now, there are small delays in the process here, typically on the order of a few milliseconds. Why so?

It turns out that these small delays allow *CSS transitions* to occur.

CSS transitions are a technology that allow you to animate a transition from one style to another. So, for example, if you changed the height of something from 10 pixels to 20 pixels, by using a CSS transition you can make the element smoothly animate to the new height. These sorts of animations are fun, often increase application usability, and are :: great mechanism to add polish and fit-and-finish to your web application.

Unfortunately, CSS transitions are not available in plain HTML: you have to use JavaScript and add or remove classes to get them to trigger. This is exactly why the htmx swap model is more complicated than you might think: by swapping in classes and adding small delays, you can access CSS transitions purely from hypermedia.

1.3.3. Taking Advantage of `htmx-swapping`

OK, so, let's go back and look at our inline delete mechanic: we click an htmx enhanced link which deletes the contact and then swaps some empty content in for the row. We know that, before the `tr` element is removed, it will have the `htmx-swapping` class added to it. We can take advantage of that to write a CSS transition that fades the opacity of the row to 0. Here is what that CSS looks like:

Listing 5. 25. Adding A Fade Out Transition

```
tr.htmx-swapping { ❶
  opacity: 0; ❷
  transition: opacity 1s ease-out; ❸
}
```

❶ We want this style to apply to `tr` elements with the `htmx-swapping` class on them

❷ The `opacity` will be 0, making it invisible

❸ The `opacity` will transition to 0 over a 1 second time period, using the `ease-out` function

Again, this is not a CSS book and I am not going to go deeply into the details of CSS transitions, but hopefully the above makes sense to you, even if this is the first time you've seen CSS transitions.

So, think about what this means from the htmx swapping model: when htmx gets content back to swap into the row it will put the `htmx-swapping` class on the row and wait a bit. This will allow the transition to a zero opacity to occur, fading the row out. Then the new (empty) content will be swapped in, which will effectively removing the row.

Sounds good, and we are nearly there. There is one more thing we need to do: the default "swap delay" for htmx is very short, a few milliseconds. That makes sense in most cases: you don't want to have much of a delay before you put the new content into the DOM. But, in this case, we want to give the CSS animation time to complete before we do the swap, we want to give it a second, in fact.

Fortunately htmx has an option for the `hx-swap` annotation that allows you to set the swap delay: following the swap type you can add `swap:` followed by a timing value to tell htmx to wait a specific amount of time before it swaps. Let's update our HTML to allow a one second delay before the swap is done for the delete action:

Listing 5. 26. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
    <a href="#" hx-delete="/contacts/{{ contact.id }}"
        hx-swap="outerHTML swap:1s" ❶
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="closest tr">Delete</a>
</td>
```

❶ A swap delay changes how long htmx waits before it swaps in new content

With this modification, the existing row will stay in the DOM for an additional second, with the `htmx-swapping` class on it. This will give the row time to transition to an opacity of zero, giving the fade out effect we want.

Now, when a user clicks on a "Delete" link and confirms the delete, the row will slowly fade out and then, once it has faded to a 0 opacity, it will be removed. Fancy! And all done in a declarative, hypermedia oriented manner, no JavaScript required! (Well, obviously htmx is written in JavaScript, but you know what I mean: we didn’t have to write any JavaScript to implement the feature!)