Hypermedia In Action

1. Improving Our Hypermedia Application

This chapter covers:

- Installing htmx in our application
- Adding AJAX-based navigation to our application
- Implementing a proper delete mechanic for contacts
- Implementing inline editing of contacts

1.1. Installing htmx

Now that we've seen how htmx extends HTML as a hypermedia, it's time to put it into action. We will still be exchanging hypermedia with our server, no need to create a JSON API, but we will be able to improve the user experience nonetheless.

The first thing we need to do is install htmx in our web application. We are going to do this by downloading the source and saving it locally in our application, so we aren't dependent on any external systems. We can grab the latest htmx version by going to `https://unpkg.com/htmx.org`, which will redirect to the current version of the library. We can copy and paste that into the `static/js/htmx.js` file. (You may want to add the version of htmx to the file name, to keep things straight.)

You can, of course, use something like Node Package Manager (NPM) or some other dependency management system if you would prefer, but we won't need a lot of javascript, and htmx is dependency free, so we'll keep it simple for our contact application.

With htmx downloaded locally to our system, we can now add the following code to the `head` tag in our `layout.html` file, so it will be included on every page in our web application:

Listing 4. 1. Installing htmx

```
    <script src="/js/htmx.js"></script>
```

That's it! No need to add a build step or anything else to our project, this simple inclusion of the htmx script file will make functionality available across our entire application.

1.2. Adding AJAX Navigation

The first feature of we are going to take advantage of is a bit of a "cheater" feature: `hx-boost`. The `hx-boost` attribute is unlike most other attributes in htmx. Other htmx attributes tend to be very focused on one aspect of HTML: `hx-trigger` focuses on the events that trigger a request, `hx-swap` focuses on how responses are swapped into the DOM, and so forth. The `hx-boost` attribute, in contrast, operates at a very high level: when you put it on an element with the value `true`, it will

"boost" all anchor tags and forms within that element. Boost, in this case, means it will convert those elements from regular anchor tags and forms into AJAX-powered anchors and forms.

So boosted anchors, for example, rather than issuing a "normal" browser request, will issue an AJAX `GET` and replace the whole body with the response.

Here is an example of a boosted link:

Listing 4. 2. A Boosted Link

```
<a href="/settings" hx-boost="true">Settings</a> ①
```

① A simple attribute makes this link AJAX-powered

Here we have a link to a hypothetical settings page. Because it has `hx-boost="true"` on it, htmx will prevent the normal link behavior and instead issue an AJAX request to `/settings`, taking the result and replacing the `body` element with the new content.

Now, you might wonder: what's the advantage here? We are issuing an AJAX request and simply replacing the entire body. Is that significantly different from just issuing a normal link request?

The answer is yes: in a boosted link, the browser is able to avoid any processing associated with the head tag. The head tag often contains many scripts and CSS file references. In the boosted scenario, it is not necessary to re-process those resources: the scripts and styles have already been processed and will continue to apply to the new content. This can often be a very easy way to speed up your hypermedia application.

A second question you might have is: does the response have to be formatted specially to work with `hx-boost`? After all, the settings page would normally render an `html` tag, with a `head` tag and so forth. Do you need to handle "boosted" requests specially?

The answer in this case is no: htmx is smart enough to pull out only the content of the body to swap in to the new page. The `head` tag, etc. are all ignored. This means you don't need to do anything special on the server side to render templates that `hx-boost` can handle: just return the normal HTML for your page and it should work fine.

Boosted form tags work in a similar way to boosted anchor tags: a boosted form will use an AJAX request rather than the usual browser-issued request, and will replace the entire body with the response:

Listing 4. 3. A Boosted Form

```
<form action="/messages" method="post" hx-boost="true">①
  <input type="text" name="message" placeholder="Enter A Message...">
  <button>Post Your Message</button>
</form>
```

① As with the link, a simple attribute makes this form AJAX-powered

This form posts messages to the `/messages` end point with an HTTP `POST`. By adding `hx-boost` to it, those requests will be done in AJAX, rather than the normal browser behavior.

Another advantage of the AJAX-based request that `hx-boost` issues is that it avoids what is known as a "flash of unstyled content", which is when a page renders before all of the styling information has been downloaded for it. This causes a disconcerting momentary flash of the unstyled content, which is then restyled when all the style information is available. You probably notice this as a flicker when you move around the internet, where text can "jump around" on the page as styles are applied to it.

With `hx-boost` the styling is already loaded before the content is retrieved, so there is no such flash of unstyled content. This makes "boosted" applications feel smoother and less jarring in general.

1.2.1. Attribute Inheritance

Let's expand on our previous example of a boosted link, and add a few more boosted links along side it:

Listing 4. 4. A Set of Boosted Links

```
<a href="/contacts" hx-boost="true">Contacts</a>
<a href="/settings" hx-boost="true">Settings</a>
<a href="/help" hx-boost="true">Help</a>
```

We now have a link to the `/contacts` page as well as the `/help` page. All these links are boosted and will behave in the manner that we have described. But this feels a little redundant, doesn't it. It is a shame we have to say `hx-boost="true"` three times here, right next to one another.

htmx offers a feature to help reduce redundancy here: attribute inheritance. For many attributes in htmx, by placing it on a parent, it will apply to all children elements. This is how Cascading Style Sheets work, and the idea was inspired by CSS.

So to avoid the redundancy in this example, lets introduce a `div` element that encloses all the links and "hoist" the `hx-boost` attribute up to it:

1. Boosting Links Via The Parent

```
<div hx-boost="true"> ①
    <a href="/contacts">Contacts</a>
    <a href="/settings">Settings</a>
    <a href="/help">Help</a>
</div>
```

① The `hx-boost` has been moved to the parent div

Now we have removed the redundant `hx-boost` attributes, but all the links are still boosted, inheriting that functionality from the parent element. Note that any legal element type could be used here, we just used a `div` out of habit.

But what if you have a link that you *don't* want boosted within an element that has `hx-boost="true"` on it? A good example is a link to a resource to be downloaded, such as a PDF. Downloading a file can't be handled well by an AJAX request, so you'd want that link to behave normally.

To deal with this situation, you would override the parent `hx-boost` value with `hx-boost="false"` on the element in question:

1. Boosting Links Via The Parent

```
<div hx-boost="true"> ①
    <a href="/contacts">Contacts</a>
    <a href="/settings">Settings</a>
    <a href="/help">Help</a>
    <a href="/help/documentation.pdf" hx-boost="false">Download Docs</a> ②
</div>
```

① The `hx-boost` is still on the parent div

② The boosting behavior is overridden for this link

Here we have a new link to a documentation PDF that we wish to function normally. We have added `hx-boost="false"` to the link and this will override the `hx-boost="true"` on the parent, reverting this link to regular link behavior and allowing the download behavior that we want.

1.2.2. Progressive Enhancement

A very nice aspect of `hx-boost` is that it "progressively enhances" web applications. Consider the links in the example above. What would happen if someone did not have JavaScript enabled? Nothing much! The application would continue to work, but it would issue regular HTTP requests, rather than AJAX-based HTTP requests. This means that your web application will work for the maximum number of users, with users of more modern browsers (or users who have not turned off JavaScript) able to take advantage of the benefits of AJAX-style navigation, but other people still able to use the app just fine.

Compare this with a JavaScript heavy Single Page Application: it simply won't function without JavaScript, obviously. It is very difficult to adopt a progressive enhancement approach within that model.

This is not to say that htmx *always* offers progressive enhancement. It is certainly possible to build features that do not offer a "No JS" fallback in htmx, and, in fact, many of the features we will build later in the book will fall into this category. (I will note when a feature is progressive enhancement friendly and when it is not.) Ultimately, it is up to you, the developer, to decide if the tradeoffs of progressive enhancement (more basic UX functionality, a limited improvement over plain HTML) are worth the benefits for your applications users.

1.2.3. Adding `hx-boost` to Contact.app

For our contact app we want this "boost" behavior… well, everywhere. Right? Why not? How could we accomplish that?

Pretty darned easy: just add `hx-boost` on the `body` tag of our `layout.html` template, and be done with it!

1. Boosting The Entire Contact.app

```
<html>
...
<body hx-boost="true">①
...
</body>
</html>
```

① All links and forms will be boosted now!

Now every link and form in our application will use AJAX by default, making it feel much snappier! All with one, single attribute. This extremely high power-to-weight ratio is why `hx-boost`, which is so different from every other attribute in htmx, is part of the library. It's just too good an idea not to include!

So, that's it, books over! You've got yourself an AJAX-powered hypermedia application now!

Of course, I'm kidding. There is a lot more to htmx, and there is a lot more room for improvement in our application, so let's keep rolling.

1.3. Deleting Contacts

In Chapter 2 you'll recall that we had a small form on the edit page of a contact to delete the contact:

Listing 4. 5. Plain HTML Form To Delete A Contact

```
    <form action="/contacts/{{ contact.id }}/delete" method="post">
        <button>Delete Contact</button>
    </form>
```

This form issued an HTTP `POST` to, for example, `/contacts/42/delete`, in order to delete the contact with the ID 42.

I mentioned previously that one of the tremendously annoying things about HTML is that you can't issue an HTTP `DELETE` (or `PUT` or `PATCH`) request directly, even though these are all part of HTTP and HTTP is *obviously designed* for transferring HTML! But now, with htmx, we have a chance to rectify this situation.

The "right thing", from a REST-ful, resource oriented perspective is, rather than issuing an HTTP `POST` to `/contacts/42/delete`, to issue an HTTP `DELETE` to `/contacts/42`. We want to delete the contact. The contact is a resource. The URL for that resource is `/contacts/42`. So the ideal situation is a `DELETE` to `` `/contacts/42/ ``.

So, how can we update our application to do this while still staying within the hypermedia model? We can simply take advantage of the `hx-delete` attribute, like so:

Listing 4. 6. An htmx Powered Button For Deleting A Contact

```
<button hx-delete="/contacts/{{ contact.id }}">Delete Contact</button>
```

Pretty simple! There are two things, in particular, to notice about this new implementation:

- We no longer need a `form` tag to wrap the button, because the button itself carries the hypermedia action that it performs directly on itself.
- We no longer need to use the somewhat awkward `"/contacts/{{ contact.id }}delete"` route, but can simply use the `"/contacts/{{ contact.id }}` route, since we are issuing a `DELETE`, which disambiguates the operation we are performing on the resource from other potential operations!

1.3.1. Updating The Server Side

Since we have updated both the route and the HTTP action we are using to delete a contact, we are going to need to update our server side implementation as well. Here is the original code:

```
@app.route("/contacts/<contact_id>/delete", methods=["POST"])
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete()
    flash("Deleted Contact!")
    return redirect("/contacts")
```

We are going to have to do two things: first we need to update the route for our handler to the new location and method we are using to delete contacts. This will be relatively straight forward.

Secondly, and this is a bit more subtle, we are going to need to change the HTTP Response Code that the handler sends back. HTTP Response Codes are numeric values that are embedded in an HTTP response that let the client know what the result of a request was. The most familiar response code for most web developers is `404`, which stands for "Not Found" and is the response code that is returned by web servers when a resource that does not exist is requested.

HTTP redirects similarly issue an HTTP Response Code, typically in the low 300s range. By default, in Flask the `redirect()` method responds with a `302` response code. According to the Mozilla Developer Network (MDN) web docs, this means that the HTTP method and body of the requests will be unchanged when the redirected request is issued. Well, in our case, we certainly don't want to issue a `DELETE` to `/contacts` when we redirect the user!

So we are going to need to update this to a `303` response code, which will convert the redirected request to a GET. Fortunately this is very easy: there is a second parameter to `redirect()` that takes the response code you wish to send.

Here is our new handler code:

```
@app.route("/contacts/<contact_id>", methods=["DELETE"]) ①
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete()
    flash("Deleted Contact!")
    return redirect("/contacts", 303) ②
```

① A slightly different path and method for the handler

② The response code is now a 303

As you can see, we removed the `/delete` ending of the path and change the method that this handler is associated with to `DELETE`. A much more natural hypermedia approach to deleting a resource!

1.3.2. Targeting The Right Element

We aren't quite out of the woods yet, however. As you may recall, by default htmx "targets" the element that triggers a request, and will place the HTML returned by the server inside that element. In this case, since the redirect to `/contacts` is going to re-render the entire contact list, we will end up in the humorous situation where the entire list ends up inside the "Delete Contact" button. Mis-targeting elements comes up from time to time in htmx and can lead to some pretty funny situations.

The fix for this is to add an explicit target to the button, targeting the `body` element with the response:

Listing 4. 7. A fixed htmx Powered Button For Deleting A Contact

```
<button hx-delete="/contacts/{{ contact.id }}"
        hx-target="body"> ①
  Delete Contact
</button>
```

① We have added an explicit target to the button now

Now our button behaves as expected: clicking on the button will issue an HTTP `DELETE` to the server against the URL for the current contact, delete the contact and redirect back to the contact list page, with a nice flash message. Perfect!

1.3.3. Updating The Location Bar URL Properly

Well, almost. If you click on the button you will notice that, despite the redirect, the URL in the location bar is not correct. It still points to `/contacts/{{ contact.id }}/delete`. This is because we haven't told htmx to update the URL. Boosting will naturally do this for you, but here we are building a custom button, and so we need to let htmx know that we want the resulting URL "pushed" into the location bar:

Listing 4. 8. Deleting A Contact, Now With Proper Location Information

```
<button hx-delete="/contacts/{{ contact.id }}"
        hx-push-url="true" ①
        hx-target="body">
  Delete Contact
</button>
```

① We tell htmx to push the redirected URL up into the location bar

Now we are done. We have a button that, all by itself, is able to issue a properly formatted HTTP `DELETE` request to the correct URL, and the UI and location bar are all updated correctly. This was accomplished with three attribute placed directly on the button, and we were able to remove the enclosing form tag as a bonus.

1.3.4. One Last Thing

And yet, if you are like me, something probably doesn't feel quite right here. Deleting a contact is a pretty darned destructive action, isn't it? And what if someone accidentally clicked on the "Delete Contact" button when they meant to click on the "Save" button?

As it stands now we would just delete that contact and too bad, so sad for the user.

Fortunately htmx has an easy mechanism for adding a confirmation message on destructive operations like this: the `hx-confirm` attribute. You can place this attribute on an element, with a message as its value, and the JavaScript method `confirm()` will be called before a request is issued, which will show a simple confirmation dialog to the user asking them to confirm the action. Very easy and a great way to prevent accidents.

Here is how we would add confirmation of the contact delete operation:

Listing 4. 9. Confirming Deletion

```
<button hx-delete="/contacts/{{ contact.id }}"
        hx-push-url="true"
        hx-confirm="Are you sure you want to delete this contact?" ①
        hx-target="body">
  Delete Contact
</button>
```

① This message will be shown to the user, asking them to confirm the delete

Now, when someone clicks on the "Delete Contact" button, they will be presented with a prompt that asks "Are you sure you want to delete this contact?" and they will have an opportunity to cancel if they clicked the button in error. Very nice.

With this final change we now have a pretty solid "delete contact" mechanic: we are using the correct, REST-ful routes and HTTP Methods, we are confirming the deletion, and we have removed a lot of the cruft that normal HTML imposes on us, all while using declarative attributes in our HTML and staying firmly within the normal hypermedia model of the web.

One thing to note about our solution, however, is that it is *not* a progressive enhancement to our web application: if someone has disabled JavaScript then this functionality will no longer work. You could do additional work to keep the older mechanism working in a JavaScript-disabled environment, but it would introduce additional and redundant code. It is up to you to determine if that tradeoff is worth the cost.

1.4. Validating Emails

A big part of any web application is validating the data that is submitted to the server side. Currently, our application has a small amount of validation that is done server side and that displays an error message when an error is detected.

We are not going to go into the details of how validation works in the model objects, but recall that the code for updating a contact looks like this:

Listing 4. 10. Server Side Validation On Contact Update

```
def contacts_edit_post(contact_id=0):
    c = Contact.find(contact_id)
    c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email'])
    if c.save(): ①
        flash("Updated Contact!")
        return redirect("/contacts/" + str(contact_id))
    else:
        return render_template("edit.html", contact=c) ②
```

① We attempt to save the contact

② If the save does not succeed we re-render the form to display error messages

And the form code in our template looks like this:

Listing 4. 11. Validation Error Messages

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="text" placeholder="Email" value="{{
contact.email }}">
    <span class="error">{{ contact.errors['email'] }}</span>①
</p>
```

① Display any errors associated with the email field

Right now there is a bit of logic in the contact class that checks if there are any other contacts with the same email, and adds an error if so: we do not want to have duplicate emails in our contacts database. This is a very common validation example: emails are usually unique and adding two contacts with the same email is almost certainly a user error.

The error message shown when a user attempts to save a contact with a duplicate email is "Email

Must Be Unique":

TODO - screen shot of application validation error

All of this is done using plain HTML and web 1.0 techniques, and it works well. However, as the application currently stands, there are two annoyances:

- First, there is no email format validation: you can enter whatever characters you'd like as an email and, as long as they are unique, the system will allow it
- Second, if a user has entered a duplicate email, they will not find this fact out until they have filled in all the fields because we only check the email's uniqueness when all the data is submitted. This could be quite annoying if the user was accidentally rentering a contact and had to put all the contact information in before being made aware of this fact!

1.4.1. Updating Our Input Type

For the first issue, we have a pure HTML mechanism for improving our application: HTML 5 supports inputs of type `email`! All we need to do is switch our input from type `text` to type `email`, and the browser will enforce that the value entered properly matches the email format:

Listing 4. 12. Changing The Input To Type `email`

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="email" placeholder="Email" value="{{
contact.email }}"> ①
    <span class="error">{{ contact.errors['email'] }}</span>①
</p>
```

① A simple change of the `type` attribute to `email` ensures that values entered are valid emails

With this change, when the user enters a value that isn't a valid email, the browser will display an error message asking for a properly formed email in that field.

So a simple single-attribute change done in pure HTML improves our validation and addresses the first annoyance we noted!

Not bad!

Server Side vs. Client Side Validations

More experienced web developers might be grinding their teeth a bit at the code above: this validation is done entirely on *the client side*. That is, we are relying on the browser to detect the malformed email and correct the user. Unfortunately, the client side is not trustworthy: a browser may have a bug in it that allows the user to circumvent the validation code. Or, worse, the user may be malicious and figure out a mechanism around our validation entirely. For example: they could simply inspect the email input and revert its type to text.

This is a perpetual danger in web development: all validations done on the client side cannot be trusted and, if the validation is important, *must be redone* on the server side. This is less of a problem in Hypermedia Driven Applications than in Single Page Applications, because the focus of HDAs is the server side, but it is still something worth bearing in mind as you build your application!

1.4.2. Inline Validation

While we have improved our validation experience a bit, the user must still submit the form to get any feedback on duplicate emails. We can use htmx to improve this user experience.

It would be better if the user were able to see a duplicate email error immediately after entering the the value. It turns out that inputs fire a "change" event and, in fact, that is the default trigger for inputs in htmx. What we want to have happen is, when the user enters an email, we immediately issue a request to the server and validate that email, then render an error message if necessary.

Recall the current HTML configuration:

Listing 4. 13. The Initial Email Configuration

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="email" placeholder="Email" value="{{
contact.email }}"> ①
    <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

So we want to add an `hx-get` to this input so that it will issue a `GET` request to a URL to validate the email. And we want to target the error span following the input with any error message returned from the server.

Listing 4. 14. Our Updated HTML

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="email"
           hx-get="/contacts/{{ contact.id }}/email" ①
           hx-target="next .error" ②
           placeholder="Email" value="{{ contact.email }}"> ①
    <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

① We issue an HTTP `GET` to the new `email` endpoint for this contact

② We target the next element with the class `error` on it, which is the next span that holds the error message

Now, with these two simple attributes in place, whenever someone changes the value of the input, an HTTP request will be issued to the given URL and, if there are errors, they will be loaded into the error span.

Now let's look at the server side implementation. We are going to add another end point, similar to our edit end point in some ways: it is going to look up the contact based on the ID encoded in the URL. In this case, however, we only want to update the email of the contact, and we obviously don't want to save it! Instead, we will call the `validate()` method on it.

That method will validate the email is unique and so forth. At that point we can return any errors associated with the email directly, or the empty string if none exist.

Here is the code:

Listing 4. 15. Our Email Validation End-Point

```
@app.route("/contacts/<contact_id>/email", methods=["GET"])
def contacts_email_get(contact_id=0):
    c = Contact.find(contact_id) ①
    c.email = request.args.get('email') ②
    c.validate() ③
    return c.errors.get('email') or "" ④
```

① Look up the contact by id

② Update its email (note that since this is a `GET`, we use the `args` property rather than the `form` property)

③ Validate the contact

④ Return a string, either the errors associated with the email field or, if there are none, the empty string

With this small bit of code in place, we now have the following very nice user experience: when a user enters an email and tabs to the next field, they are immediately notified if the email is already taken!

Here again I want to stress that this interaction is done entirely within the hypermedia model: we are using declarative attributes to exchange hypermedia with the server in a manner very similar to how links or forms work, but we have managed to improve our user experience dramatically!

Note that the email validation is *still* done when the entire contact is submitted for an update, so there is no danger of allowing duplicate email contacts to slip through: we have simply made it possible for users to catch this situation earlier by use of htmx.

It is also worth noting that this email validation *must* be done on the server side: you cannot determine that an email is unique across all contacts unless you have access to the data store of record. This is another simplifying aspect of Hypermedia Driven Applications: since validations are done server side, you have access to all the data you might need to do any sort of validation you'd like.

1.4.3. Going Further

Now, despite the fact that we haven't written a lot of code here, this is a fairly sophisticated user interface, at least when compared with plain HTML-based applications. However, if you have used more advanced web applications you have probably seen the pattern where an email field (or similar) is validated *as you type*.

This is surely beyond the reach of a Hypermedia Driven Application, right? Only a sophisticated Single Page Application framework could provide that level of interactivity!

Oh ye of little faith. With a bit more effort, we can use htmx to achieve this user experience.

In fact, all we need to do is to change our trigger. Currently, we are using the default trigger for inputs, which is the `change` event. To validate as the user types, we would want to capture the `keyup` event as well:

Listing 4. 16. Triggering With `keyup` Events

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="email"
           hx-get="/contacts/{{ contact.id }}/email"
           hx-target="next .error"
           hx-trigger="change, keyup" ①
           placeholder="Email" value="{{ contact.email }}"> ①
    <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

① An explicit trigger has been declared, and it triggers on both the `change` and `keyup` events

With this tiny change, every time a user types a character we will issue a request and validate the email! Simple!

Unfortunately, this is probably not what you want: issuing a new request on every key up event would be very wasteful and could potentially overwhelm your server. What we want to do is only issue the request if the user has paused for a small amount of time. This is called "debouncing" the

input, where requests are delayed until things have "settled down".

htmx supports a `delay` modifier for triggers that allows you to debounce a request by adding a delay before the request is sent. If another event of the same kind appears within that interval, htmx will not issue the request and will reset the timer. This is exactly what we want for this situation. Let's add a delay of 200 milliseconds to the `keyup` trigger:

Listing 4. 17. Debouncing the `keyup` Event

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="email"
           hx-get="/contacts/{{ contact.id }}/email"
           hx-target="next .error"
           hx-trigger="change, keyup delay:200ms" ①
           placeholder="Email" value="{{ contact.email }}">
    <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

① We debounce the `keyup` event by adding a `delay` modifier

Now we no longer issue a stream of validation requests as the user types. Instead, we wait until the user pauses for a bit and then issue the request. Much better for our server, and still a great user experience!

There is one last thing we might want to address: as it stand we will issue a request no matter *which* keys are pressed, even if they are keys like the arrow keys, which have no effect on the value of the input. It would be nice if there were a way to only issue a request if the input value has changed. It turns out that htmx has support for that pattern using the `changed` modifier for events. (Not to be confused with the `change` event!)

By adding `changed` to our `keyup` trigger, the input will not issue validation requests unless the keyup event actually updates the inputs value:

Listing 4. 18. Only Sending Requests When The Input Value Changes

```
<p>
    <label for="email">Email</label>
    <input name="email" id="email" type="email"
           hx-get="/contacts/{{ contact.id }}/email"
           hx-target="next .error"
           hx-trigger="change, keyup delay:200ms changed" ①
           placeholder="Email" value="{{ contact.email }}">
    <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

① We do away with pointless requests by only issuing them when the inputs value has actually changed

Now that's some pretty good-looking code! With a total of three attributes and a simple new server-side end point, we have added a fairly sophisticated user experience to our web application. Even

better, any email validation rules we add on the server side will *automatically* just work using this model: because we are using hypermedia as our communication mechanism there is no need to keep a client-side and server-side model in sync with one another.

This is a great demonstration of the power of the hypermedia architecture!

1.5. Paging

Currently, our application does not support paging: if there are 100 contacts in the database we will show 100 contacts on the main page. Let's fix that, so that we only show ten contacts at a time with a "Next" and "Previous" link if there are more than 10 or if we are beyond the first page.

The first change we will need to make is to add a simple paging widget to our `index.html` template. Here we will conditionally include two links:

- If we are beyond the first page, we will include a link to the previous page
- If there are ten contacts in the current result set, we will include a link to the next page

This isn't a perfect paging widget: ideally we'd show the number of pages and offer the ability to do more specific page navigation, and there is the possibility that the next page might have 0 results in it since we aren't checking the total results count, but it will do for now for our simple application.

Let's look at the jinja template code for this.

Listing 4. 19. Adding Paging Widgets To Our List of Contacts

```
<div>
    <span  style="float: right"> ①
        {% if page > 1 %}
          <a href="/contacts?page={{ page - 1 }}">Previous</a> ②
        {% endif %}
        {% if contacts|length == 10 %}
          <a href="/contacts?page={{ page + 1 }}">Next</a> ③
        {% endif %}
    </span>
</div>
```

① Include a new div under the table to hold our navigation links

② If we are beyond page 1, include an anchor tag with the page decremented by one

③ If there are 10 contacts in the current page, include an anchor tag linking to the next page by incrementing it by one

Note that here we are using the special jinja syntax `contacts|length` to compute the length of the contacts list.

Now lets address the server side implementation.

We need to look for the `page` parameter and pass that through to our model as an integer so the model knows what page of contacts to return:

Listing 4. 20. Adding Paging To Our Request Handler

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    page = int(request.args.get("page", 1)) ①
    if search:
        contacts_set = Contact.search(search)
    else:
        contacts_set = Contact.all(page) ②
    return render_template("index.html", contacts=contacts_set, page=page)
```

① Resolve the page parameter, defaulting to page 1 if no page is passed in

② Pass the page through to the model when loading all contacts so it knows which page of 10 contacts to return

This is fairly straightforward: we just need to get another parameter, like the `q` parameter we passed in for searching contacts earlier, convert it to an integer and then pass it through to the `Contact` model so it knows which page to return.

And that's it. We now have a very basic paging mechanism for our web application. And, believe it or not, it is already using AJAX, thanks to our use of `hx-boost` in the application. Easy!

1.5.1. Click To Load

Now, the current paging mechanism is fine, although it could use some additional polish. But sometimes you don't want to have to page through items and lose your place in the application. In cases like this a different UI pattern might be better. For example, you may want to load the next page *inline* in the current page. This is the common "click to load more" UX pattern.

Let's see how we can implement this in htmx.

It's actually surprisingly simple: we can just take the existing "Next" link and repurpose it a bit using nothing but htmx attributes!

We want to have a button that, when clicked, appends the rows from the next page of contacts to the current, exiting table, rather than re-rendering the whole table. This can be achieved by adding a row to our table that has just such a button in it:

Listing 4. 21. Changing To "Click To Load"

```
        <tbody>
        {% for contact in contacts %}
            <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 %}
        {% if contacts|length == 10 %} ①
            <tr>
                <td colspan="5" style="text-align: center">
                    <button hx-target="closest tr" ②
                            hx-swap="outerHTML"   ③
                            hx-select="tbody > tr" ④
                            hx-get="/contacts?page={{ page + 1 }}">Load More</button>
                </td>
            </tr>
        {% endif %}
        </tbody>
```

① As with the "Next" link in our paging example, we only show "Load More" if there are 10 contact results in the current page

② In this case, the button needs to target the closest enclosing row, which is what the `closest` syntax allows

③ We want to replace this row with the response from the server

④ Of course, we don't want to replace the row with the entire response, we only want to replace it with the rows within the table body of the response, so we use the `hx-select` attribute to select those rows out using a standard CSS selector

Believe it or not, that's all we need to change to enable a "Click To Load" style UI! No server side changes are necessary because of the flexibility that htmx gives you with respect to how we process server responses. Pretty cool, eh?

1.5.2. Infinite Scroll

Another somewhat common pattern for dealing with long lists of things is known as "infinite scroll", where, as the end of a list or table is scrolled into view, more elements are loaded. This behavior makes more sense in situations where a user is exploring a category or series of social media posts, rather than in the context of a contact application, but for completeness we will show how to achieve this in htmx.

We can repurpose the "Click To Load" code to implement this new pattern. If you think about it for a moment, really infinite scroll is just the "Click To Load" logic, but rather than loading when a click occurs, we want to load when an element is "revealed" in the view portal of the browser.

As luck would have it, htmx offers a synthetic (non-standard) DOM event, `revealed` that can be used in tandem with the `hx-trigger` attribute, to trigger a request when, well, when an element is revealed. Let's convert our button to a span and take advantage of this event:

Listing 4. 22. Changing To "Click To Load"

```
{% if contacts|length == 10 %} ①
    <tr>
        <td colspan="5" style="text-align: center">
            <span hx-target="closest tr" ①
                  hx-trigger="revealed" ②
                  hx-swap="outerHTML"
                  hx-select="tbody > tr"
                  hx-get="/contacts?page={{ page + 1 }}">Loading More...</span>
        </td>
    </tr>
{% endif %}
```

① We have converted our element from a button to a span, since the user will not be clicking on it

② We trigger the request when the element is revealed, that is when it comes into view in the portal

So all we needed to do to convert from "Click to Load" to "Infinite Scroll" was update our element to be a span and add the `revealed` trigger. The fact that this was so easy shows how well htmx generalizes HTML: just a few attributes allow us to dramatically expand what we can achieve with our hypermedia. And, again, I note that we are doing all this within the original, REST-ful model of the web, exchanging hypermedia with the server. As the web was designed!

1.6. Summary

- In this chapter we began improving our Hypermedia-Driven Application (HDA) by using the htmx library
- A simple and quick way to improve the application was to use the `hx-boost` attribute, which "boosts" all links and forms to use AJAX interactions
- Deleting a contact could be updated to use the proper `DELETE` HTTP request, using the `hx-delete` attribute
- Validating the email of a contact as the user entered it was achieved using a combination of `hx-get` and `hx-target`
- Paging was added to the application using standard server-side techniques
- We implemented the "Click To Load" pattern using a simple button and a few htmx attributes, without any server side changes
- We repurposed the "Click To Load" example to implement "Infinite Scroll" by adding a single attribute, `hx-trigger` with the value `reveal`