Hypermedia In Action

11. Creating A Dynamic Archive UI

This chapter covers

- Creating a dynamically updated download UI using hypermedia
- Utilizing declarative polling to update content on an interval
- Adding smooth animations to a progress bar
- Triggering a file download with a response header

A Dynamic Archive UI

We've come a long way from a traditional web 1.0-style web application at this point: we've added active search, bulk delete (with some nice animations) and a slew of other features. We have also built a HyperView-based mobile application, taking advantage of that mobile-native hypermedia. Overall, we have reached a level of interactivity that most people would assume requires some sort JavaScript framework for our application, but we've done nearly all of it with good old hypermedia, plus a bit of hypermedia-compatible scripting on the side.

Let's look at how we can add a final, major feature to Contact.app: downloading an archive of our contacts.

From a hypermedia perspective, downloading a file isn't exactly rocket science: using the HTTP `Content-Disposition` response header, we can easily tell the browser to download and save a file to the local computer. However, let's make this problem a bit more interesting: let's add in the fact that the export can take a bit of time, from five to ten seconds, or sometimes even longer, to complete.

This means if we implemented it as a normal HTTP request, driven by a link or a button, the user might sit, with very little visual feedback, wondering if the download is actually happening, while the export is being run. They might eve n click the download control again, causing a *second* archive request. Not good!

This is a classic problem in web app development. When faced with potentially long-running process like this, we ultimately have two options:

- When the user triggers the action, block until it is complete and then respond with the result
- Start the action and return immediately

Just blocking and waiting for the action to complete is certainly the easy way to handle it, but it is a pretty terrible user experience. If you've ever clicked on something in a web 1.0-style application and then had to sit there for what seems like an eternity before anything happens, you've seen the results of this choice.

The second option, starting the action asynchronously (say, by creating a thread, or submitting it to a job runner system) is much nicer from a user experience perspective: you can respond immediately and the user doesn't need to sit there wondering what's going on.

But the question is, what do you respond *with*? The job probably isn't complete yet, so you can't just provide a link to the results, can you?

I have seen a few different "simple" approaches in this scenario:

- Let the user know that the process has started and that they will be emailed a link to the completed process results when it is finished
- Let the user know that the process has started and recommend that they should manually (!!!) refresh the page to see the status of the process
- Let the user know that the process has started and automatically refresh the page every few seconds using some JavaScript

All of these work. But they all are a somewhat awkward user experience, are they?

What we'd *really* like in this scenario is something more like what you see when, for example, you download a large file via the browser: a nice progress bar indicating where in the process you are, and, when the process is complete, a link to click immediately to view the result of the process.

This may sound like something impossible to implement with hypermedia, and, frankly, we'll need to push htmx pretty hard to make this all work, but, when it is done, it won't be *that* much code, and we will be able to achieve the user experience we want for this archiving feature.

Let’s get into it.

UI Requirements

Before we dive into the implementation, let’s discuss in broad terms what our new UI should look like: we want a button in the application labeled "Download Contact Archive". When a user clicks on that button, we want to replace that button with a progress bar. As the archive job progresses, we want to move the progress bar along. When the archive job is done, we want to show a link to the user to download the archive file.

In order to actually do the archiving, we are going to use a python class, `Archiver`, that implements all the functionality that we need. We aren’t going to go into the implementation details of the class, because that’s beyond the scope of this book. Let’s say that it was created by another developer, and it provides all the server-side behavior necessary to produce a contact archive.

In particular, it gives us the following methods to work with:

- `status()` - A string representing the status of the download, either `Waiting`, `Running` or `Complete`
- `progress()` - A number between 0 and 1, indicating how much progress the archive job has made
- `run()` - Starts a new archive job (if the current status is `Waiting`)
- `reset()` - Cancels the current archive job, if any, and resets to the "Waiting" state
- `archive_file()` - The path to the archive file that has been created on the server, so we can send it to the client
- `get()` - A class method that lets us get the Archiver for the current user

This is not a terribly complicated API, is it? The only somewhat tricky aspect to the whole API is that the `run()` method is *non-blocking*. This means that it does not *immediately* create the archive file, but rather it starts a background job (as a thread) to do the actual archiving. This can be confusing if you aren’t used to multithreading in code: you might be expecting the `run()` method to "block", that is, to actually execute the entire export and only return when it is finished. But, if it did that, we wouldn’t be able to start the archive process and immediately render our desired archive progress UI, would we?

Beginning Our Implementation

We now have everything we need to begin implementing our UI: a reasonable outline of what it is going to look like, and the domain logic to support it.

So, in getting down to building the UI, the first thing I want to note is that the UI is largely self-contained: we want to replace the button with the download progress bar, and then the link to download the results of the archive process. The fact that our archive user interface is all going to be within a specific part of the UI is a strong hint that we want to create a new template to handle it. Let's call this template `archive_ui.html`.

Another thing that jumps out at me is that we are going to want to replace the entire download UI in multiple cases:

- When we start the download, we will want to replace the button with a progress bar
- As the archive process proceeds, we will want to replace/update the progress bar
- When the archive process completes, we will want to replace the progress bar with a download link

Given we are going to be updating the UI in this way, it makes sense to have a good target for the updates. So, let's wrap the entire UI in a `div` tag, and then use that `div` as the target for all our operations.

Here is the start of the template content for our new archive user interface:

Listing 11. 1. Our Initial Archive UI Template

```
<div id="archive-ui" hx-target="this"<1> hx-swap="outerHTML"<2>>

</div>
```

❶ This div will be the target for all elements inside of it

❷ Replace the entire div every time using `outerHTML`

Next, lets add that "Download Contact Archive" button to the `div`, which will kick off the archive-then-download process. Let's use a `POST` to the path `/contacts/archive` to trigger the start of the process:

Listing 11. 2. Adding The Button

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
  <button hx-post="/contacts/archive"> ❶
      Download Contact Archive
  </button>
</div>
```

❶ This button will issue a `POST` to `/contacts/archive`

Finally, let's include this template in our main `index.html` template, above the contacts table:

Listing 11. 3. Our Initial Archive UI Template

```
{% block content %}

    {% include 'archive_ui.html' %} ❶

    <form action="/contacts" method="get" class="tool-bar">
```

❶ This template will now be included in the main template

With that done, we now have a button showing up in our web application to get the download going. Since the enclosing `div` has an `hx-target="this"` on it, the button will inherit that target and replace the `div` with whatever HTML comes back from the `POST` to `/contacts/archive`.

Adding the POST End Point

Our next step is to handle the `POST` that the button is making. What we are going to want to do is to get the `Archiver` for the current user and invoke the `run()` method on it. This will start the archive process running. Then we will want to render some new content indicating that the process is running.

To do that, what we want to do is reuse the `archive_ui` template to handle rendering the archive UI for both states, when the archiver is "Waiting" and when it is "Running". (We will handle the "Complete" state in a bit.)

This is a very common pattern: we put all the different potential UIs for a given chunk of the user interface into a single template, and conditionally render the appropriate interface.

By keeping everything in one file, it makes it much easier for other developers (or for us, if we come back after a while!) to understand exactly how the UI works on the client side.

Since we are going to conditionally render different user interfaces based on the state of the archiver, we will need to pass the archiver out to the template as a parameter. So, again: we need to invoke `run()` on the archiver in our controller and then pass the archiver along to the template, so it can render the UI appropriate for the current status of the archive process.

Here is what the code looks like:

Listing 11. 4. Server Side Code To Start The Archive Process

```
@app.route("/contacts/archive", methods=["POST"]) ❶
def start_archive():
    archiver = Archiver.get() ❷
    archiver.run() ❸
    return render_template("archive_ui.html", archiver=archiver) ❹
```

❶ Handle `POST` to `/contacts/archive`

❷ Look up the Archiver

❸ Invoke the non-blocking `run()` method on it

❹ Render the `archive_ui.html` template, passing in the archiver

Conditionally Rendering A Progress UI

Now let's turn our attention to updating `archive_ui.html` to conditionally. We are passing the archiver through as a variable to the template, and recall that the archiver has a `status()` method that we can consult to see what the status of the archive process.

We want to render the "Download Contact Archive" button if the archiver has the status `Waiting`, and we want to render some sort of message indicating that progress is happening if the status is `Running`. Let's update our template code to do just that:

Listing 11. 5. Adding Conditional Rendering

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
    {% if archiver.status() == "Waiting" %} ❶
        <button hx-post="/contacts/archive">
            Download Contact Archive
        </button>
    {% elif archiver.status() == "Running" %}❷
       Running...❸
    {% end %}

</div>
```

❶ Only render button if the status is "Waiting"

❷ Render different content when status is "Running"

❸ For now, just some text saying things are Running

OK, great, we have some conditional logic in our template view, and the server side logic to support kicking off the archive process. We don't have a progress bar yet, but we'll get there! Let's see how this works as it stands, and refresh the main page of our application…

Listing 11. 6. Something Went Wrong

```
UndefinedError
jinja2.exceptions.UndefinedError: 'archiver' is undefined
```

Ouch!

We get an error message right out of the box. Why? Ah, of course, we are including the `archive_ui.html` in the `index.html` template, but now the `archive_ui.html` template expects the archiver to be passed through to it, so it can conditionally render the correct UI. Well, that's an easy fix: we just need to pass the archiver through when we render the `index.html` template as well:

Listing 11. 7. Including The Archiver When We Render index.html

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q")
    if search is not None:
        contacts_set = Contact.search(search)
        if request.headers.get('HX-Trigger') == 'search':
            return render_template("rows.html", contacts=contacts_set)
    else:
        contacts_set = Contact.all()
    return render_template("index.html", contacts=contacts_set,
archiver=Archiver.get())❶
```

❶ Pass through archiver to the main template

Now with that done, we can load up the page. And, sure enough, we can see the "Download Contact Archive" button now! When we click on it, the button is replaced with the content "Running…", and we can see in our development console on the server side that the job is indeed getting kicked off properly.

11.1. Polling

That’s definitely progress, but we don’t exactly have the best progress indicator here: just some static text telling the user that the process is running!

What we want to do is have the content update as the process makes progress and, ideally, show a progress bar indicating how far along it is. How can we do that in htmx using plain old hypermedia?

The technique we want to use here is called "polling", where we issue a request on an interval and update the UI based on the new state of the server.

Polling? Really?

Polling has a bit of a bad rap, and it isn't the sexiest technique in the world: today developers might look at a more advanced technique like WebSockets or Server Sent Events (SSE) to address this situation.

But, say what one will, polling *works* and it is drop-dead simple. You need to be careful to make sure you don't overwhelm you system with polling requests, but, with a bit of care, you can create a reliable, passively updated component in your UI.

htmx offers two types of polling. The first is "fixed rate polling", which uses a special `hx-trigger` syntax to indicate that something should be polled on a fixed interval.

Here is an example:

Listing 11. 8. Fixed Interval Polling

```
<div hx-get="/messages" hx-trigger="every 3s"> ❶
</div>
```

❶ trigger a `GET` to `/messages` every three seconds

This works great in situations when you want to poll indefinitely, for example if you want to constantly poll for new messages to display to the user. However, fixed rate polling isn't ideal when you have a definite process after which you want to stop polling: it keeps polling forever, until the element it is on is removed from the DOM.

In our case, we have a definite process with an ending to it. So, in our case, it will be better to use the other polling technique, known as "load polling". In load polling, you take advantage of the fact that htmx triggers a `load` event when content is loaded into the DOM. So you can create a trigger on the `load` event, but then add a bit of a delay so that the request doesn't trigger immediately.

If you do this, then you can conditionally render the `hx-trigger` on every request: when a process has completed you can simply not include the trigger and the load polling stops. A nice and simple way to poll for until a definite process finishes.

11.1.1. Using Polling To Update The Archive UI

So, let's use load polling now to update our UI as the archiver makes progress. To show the progress, let's use a CSS-based progress bar, taking advantage of the `progress()` method which returns a number between 0 and 1 indicating how close the archive process is to completion. Here is the snippet of HTML we will use:

Listing 11. 9. A CSS-based Progress Bar

```
<div class="progress" >
    <div class="progress-bar" style="width:{{ archiver.progress() * 100 }}%"></div>
❶
</div>
```

❶ The width of the inner element corresponds to the progress

This CSS-based progress bar has two components: an outer `div` that provides the wire frame for the progress bar, and and inner `div` that is the actual progress bar indicator. We set the width of the inner progress bar to some percentage (note we need to multiply the `progress()` result by 100 to get a percentage) and that will make the progress indicator the appropriate width within the parent div.

As I have mentioned before, this is not a book on CSS, but, for completeness, here is the CSS for this progress bar:

Listing 11. 10. The CSS For Our Progress Bar

```
.progress {
    height: 20px;
    margin-bottom: 20px;
    overflow: hidden;
    background-color: #f5f5f5;
    border-radius: 4px;
    box-shadow: inset 0 1px 2px rgba(0,0,0,.1);
}

.progress-bar {
    float: left;
    width: 0%;
    height: 100%;
    font-size: 12px;
    line-height: 20px;
    color: #fff;
    text-align: center;
    background-color: #337ab7;
    box-shadow: inset 0 -1px 0 rgba(0,0,0,.15);
    transition: width .6s ease;
}
```

Which ends up rendering like this:

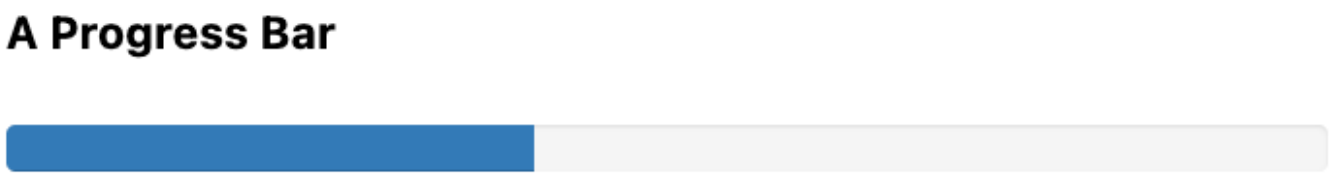

Figure 11. 1. Our CSS-Based Progress Bar

So let's add the code for our progress bar into our `archive_ui.html` template for the case when the archiver is running, and let's update the copy to say "Creating Archive…":

Listing 11. 11. Adding The Progress Bar

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
    {% if archiver.status() == "Waiting" %}
        <button hx-post="/contacts/archive">
            Download Contact Archive
        </button>
    {% elif archiver.status() == "Running" %}
        <div>
            Creating Archive...
            <div class="progress" > ❶
                <div class="progress-bar" style="width:{{ archiver.progress() * 100
}}%"></div>
            </div>
        </div>
    {% endif %}
</div>
```

❶ Our shiny new progress bar

Sweet, now when we click the "Download Contact Archive" button, we get the progress bar. But it still doesn't update because we haven't implemented load polling yet! It just sits there, at zero.

To get the UI we want, we'll need to implement load polling using `hx-trigger`. We can add this to pretty much any element inside the conditional block for when the archiver is running, so let's add it to that `div` that is wrapping around the "Creating Archive…" text and the progress bar. Finally, let's make it poll by issuing a `GET` to the same path that the `POST` was issued too: `/contacts/archive`. (As you have probably notices, this is a common pattern in RESTful systems: reusing the same path with different actions.)

Listing 11. 12. Implementing Load Polling

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
    {% if archiver.status() == "Waiting" %}
        <button hx-post="/contacts/archive">
            Download Contact Archive
        </button>
    {% elif archiver.status() == "Running" %}
        <div hx-get="/contacts/archive" hx-trigger="load delay:500ms"> ❶
            Creating Archive...
            <div class="progress" >
                <div class="progress-bar" style="width:{{ archiver.progress() * 100
}}%"></div>
            </div>
        </div>
    {% endif %}
</div>
```

❶ Issue a `GET` to `/contacts/archive` 500 milliseconds after the content loads

Again, it is important to realize that, when this `GET` is issued to `/contacts/archive`, it is going to replace the `div` with the id `archive-ui`, not just itself. The `hx-target` attribute is *inherited* by all child elements within the `archive-ui div`, so, unless it is explicitly overriden by a child, the children will all target that outermost `div` in the `archive_ui.html` file.

OK, now we need to handle the `GET` to `/contacts/archive` on the server. Thankfully, this is quite easy: all we want to do is re-render `archive_ui.html` with the archiver:

Listing 11. 13. Handling Progress Updates

```
@app.route("/contacts/archive", methods=["GET"]) ❶
def archive_status():
    archiver = Archiver.get()
    return render_template("archive_ui.html", archiver=archiver) ❷
```

❶ handle `GET` to the `/contacts/archive` path

❷ just re-render the `archive_ui.html` template

Simple, like so much else with hypermedia!

And now, when we click the "Download Contact Archive", sure enough, we get a progress bar that updates every 500 milliseconds! And, as the result of the call to `archiver.progress()` incrementally updates from 0 to 1, the progress bar moves across the screen for us, very cool!

11.1.2. Downloading The Result

OK, we have one more state to handle, the case when `achiver.status()` is set to "Complete", and there is a JSON archive of the data ready to download. When the archiver is complete, we can get the local JSON file on the server from the archiver via the `archive_file()` call.

Let's add another case to our if statement to handle the "Complete" state, and, when the archive job is complete, lets render a link to a new path, `/contacts/archive/file`, which will respond with the archived JSON file. Here is the new code:

Listing 11. 14. Rendering A Download Link When Archiving Completes

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
    {% if archiver.status() == "Waiting" %}
        <button hx-post="/contacts/archive">
            Download Contact Archive
        </button>
    {% elif archiver.status() == "Running" %}
        <div hx-get="/contacts/archive" hx-trigger="load delay:500ms">
            Creating Archive...
            <div class="progress" >
                <div class="progress-bar" style="width:{{ archiver.progress() * 100
}}%"></div>
            </div>
        </div>
    {% elif archiver.status() == "Complete" %} ❶
        <a hx-boost="false" href="/contacts/archive/file">Archive Ready!  Click here
to download. &downarrow;</a> ❷
    {% endif %}
</div>
```

❶ If the status is "Complete", render a download link

❷ The link will issue a `GET` to `/contacts/archive/file`

Note that the link has a `hx-boost` set to `false`. It has this so that the link will not inherit

the boost behavior that is present for other links and, thus, will not be issued via AJAX. We want this "normal" link behavior because an AJAX request cannot download a file directly, whereas a plain anchor tag can.

11.1.3. Downloading The Completed Archive

The final step is to handle the `GET` request to `/contacts/archive/file`. We want to send the file that the archiver created down to the client. We are in luck: flask has a very simple mechanism for sending a file as a downloaded response: the `send_file()` method.

We can pass this method the path to the archive file that the archiver created, the name of the file that we want the browser to create, and if we want it sent "as an attachment". This last argument will which will tell Flask to set the HTTP response header `Content-Disposition` to `attachment` with the given filename, which will trigger the browsers file-downloading behavior.

Listing 11. 15. Sending A File To The Client

```
@app.route("/contacts/archive/file", methods=["GET"])
def archive_content():
    manager = Archiver.get()
    return send_file(manager.archive_file(), "archive.json", as_attachment=True) ❶
```

❶ send the file to the client

Perfect! Now we have an archive UI that is pretty darned slick: You can click the button and a progress bar appears. When the progress bar reaches 100%, it disappears and a link to download the archive file appears. The user can then click on that link and download their archive!

11.2. Smoothing Things Out: Animations in htmx

As cool as this UI is, there is one minor annoyance with it: as the progress bar updates it "jumps" from one position to the next. This looks jerky and is reminiscent of the feel of a full page refresh in web 1.0 style applications. It turns out that there is a native HTML technology for smoothing out changes on an element from one state to another that we discussed in Chapter 5: the CSS Transitions API.

Using CSS Transitions, you can smoothly animate an element between different styling by using the `transition` property.

If you look back at our CSS definition of the `.progress-bar` class, you will see the following transition definition in it: `transition: width .6s ease;`. This means that when the width of the progress bar is changed from, say 20% to 30%, the browser will animate over a period of .6 seconds using the "ease" function (which has a nice accelerate/decelerate effect).

That's great and all, but in our example, htmx is *replacing* the content with new content. It isn't updating the width of the *existing* element, which would trigger a transition. Rather, it is simply replacing it with a new element. So no transition will occur, which is, indeed, what we are seeing: the progress bar jumps from spot to spot as it moves towards completion.

11.2.1. The "Settling" Step in htmx

When we discussed the htmx swap model in Chapter 5, we focused on the classes that htmx adds and removes, but we skipped over the idea of "settling". What is "settling" in htmx terms? Settling is the following process: when htmx is about to replace a chunk of content, it looks through the new content and finds all elements with an `id` on it. It then looks in the *existing* content for elements with the same `id`. If there is one, it does the following shuffle:

- The *new* content gets the attributes of the *old* content temporarily
- The new content is inserted
- After a small delay, the new content has its attributes reverted to their actual values

So, what is this strange little dance supposed to achieve? Well, what this ends up meaning is that, if an element has a stable id between swaps, you *can* write CSS transitions between various states. Since the new content briefly has the *old* attributes, the normal CSS mechanism will kick in when the actual values are restored.

So, in our case, all we need to do is to add a stable ID to our `progress-bar` element, and, rather than jumping on every update, it the progress bar should smoothly move across the screen as it is updating, using the CSS transition defined in our style sheet:

Listing 11. 16. Smoothing Things Out

```
<div class="progress" >
    <div id="archive-progress" class="progress-bar" style="width:{{
archiver.progress() * 100 }}%"></div> ❶
</div>
```

❶ The progress bar div now has a stable id across requests

So, despite all the complicated mechanics going on behind the scenes in htmx, all we have to do, as an htmx user, is add a simple `id` attribute to the element we want to animate. And, viola, nice, smooth progress bar, even though we are replacing the content with new HTML!

11.3. Dismissing The Download UI

Next, let's make it possible for the user to dismiss the download link and return to the original export UI state. To do this, we'll add a button that issues a `DELETE` to the path `/contacts/archive`, indicating that the current archive can be removed or cleaned up.

We'll add it after the download link, like so:

Listing 11. 17. Clearing The Download

```
    <a hx-boost="false" href="/contacts/archive/file" _="on load click() me">Archive
Ready!  Click here to download. &downarrow;</a>
    <button hx-delete="/contacts/archive">Clear Download</button> ❶
```

❶ A simple button that issues a `DELETE` to `/contacts/archive`

Now the user has a button that they can click on to dismiss the archive download link. But we will need to hook it up on the server side. As usual, this is pretty straight forward: we simply create a new handler for the `DELETE` HTTP Action, invoke the `reset()` method on the archiver, and re-render the `archive_ui.html` template. Since this button is picking up the same `hx-target` and `hx-swap` configuration as everything else, it "just works".

Here is the server side code:

Listing 11. 18. Resetting The Download

```
@app.route("/contacts/archive", methods=["DELETE"])
def reset_archive():
    archiver = Archiver.get()
    archiver.reset() ❶
    return render_template("archive_ui.html", archiver=archiver)
```

❶ Call `reset()` on the archiver

Looks pretty similar to our other methods, doesn't it? That's the idea!

11.4. An Alternative UX: Auto-Download

While I prefer the current user experience for archiving contancts, where a progress bar shows the progress of the process and, when it completes, I am given a new hypermedia control to actually download the file, there are other alternatives to it. Another pattern that I see on the web is "auto-downloading": rather than requiring the user to click an additional button or link when the archive (or other process) is complete, instead the system automatically downloads the file without any additional user interaction.

We can add this functionality quite easily to our application with just a bit of scripting. We will use hyperscript because it is our preferred scripting option, but the equivalent JavaScript should be obvious.

All we need to do to implement this auto-download feature is the following: when the download link renders, simply automatically click on the link for the user.

The hyperscript will read basically just like that sentence:

Listing 11. 19. Auto-Downloading

```
<a hx-boost="false" href="/contacts/archive/file"
   _="on load click() me"> ❶
  Archive Downloading!  Click here if the download does not start.
</a>
```

❶ a bit of hyperscript to make the file auto-download

Have we mentioned that we like hyperscript?

Note, once again, that the scripting here is simply *enhancing* the existing hypermedia,

rather than replacing it with a non-hypermedia request. This is hypermedia-friendly scripting!

So, despite our initial trepidation that it could be done, we've managed to create a very dynamic UI for our archive functionality, with a progress bar and auto-downloading, and we've done nearly all of it (with the exception of a small bit of scripting for auto-download) in pure hypermedia. And it only took about 16 lines of front end code and 16 lines of backend code to build the whole thing, showing once again that HTML, with the help of htmx, can, in fact, be very expressive.

11.5. Summary

- In this chapter we built a sophisticated user interface to interact with a non-blocking, asynchornous back end process
- We saw a two different ways to do polling in htmx, both "indefinite" polling using the `hx-trigger` attribute and a technique called "load polling"
- We settled on using "load polling" for our situation, since it had a definite end point after which we no longer wanted to poll
- We saw how the htmx swap mechanism enables CSS transitions when an element has a stable ID in new pieces of content, and we used that to smooth out the progress bar in our application
- We used a bit of hypermedia-friendly scripting to trigger an auto-download when the archive progress completes