Hypermedia In Action

1. ContactApp

This chapter covers:

- Building a simple contact management web application
- Server Side Rendering (SSR) with HTML

1.1. A Contact Management Web Application

To begin our journey into Hypermedia Driven Applications, we are going to create a simple contact management web application named Contacts.app. We will start with a basic, Web 1.0-style multi-page application, in the grand CRUD (Create, Read, Update, Delete) tradition. It will not be the best contact management application, but that's OK because it will be simple (a great virtue of web 1.0 applications!) and it will be easy to incrementally introduce htmx to improve the application and get it closer to a polished, modern web application.

By the time we are finished with the application it will have some very slick features that most developers today would assume requires sophisticated client-side infrastructure. We will, instead, implement these features entirely using hypermedia mixed with a bit of light client side scripting.

1.1.1. Which Stack?

In order to demonstrate how a hypermedia application works, we need to pick a server-side language and library for handling HTTP requests. Colloquially, this is called our "server sid stack", and there are hundreds of options out there, many with passionate followers. For our application we are going to go with the following: Python and Flask, with Jinja2 templates.

Why pick this stack? Well, we picked Python because it is the most popular programming language today, and even if you don't know or like Python, it is very easy to read.

We picked Flask because it does not impose a lot of structure on top of the basics of HTTP routing. This bare bones approach isn't for everyone: many people prefer the "Batteries Included" nature of Django, for example. We understand that, but for demonstration purposes, we feel that an unopionated and light-weight library will make it easier for non-Python developers to follow along, and anyone who prefers django or some other Python web framework, or some other language entirely, should be able to easily convert the Flask examples into their native framework.

Flask uses Jinja2 templates, which are simple enough and standard enough that most people who understand any server side (or client side) templating library will be able to pick them up quickly and easily. We will intentionally keep things simple (sometimes sacrificing other design principles to do so!) to maximize the teaching value of our code: it won't be perfectly factored code, but it will be easy enough to follow for the majority of people interested in web development.

The HOWL Stack: Hypermedia On Whatever you'd Like

We picked Python and Flask for this book, but we could have picked anything. One of the wonderful things about building a hypermedia-based application is that your backend can be… whatever you'd like!

If we were building a web application with a large JavaScript-based front end application, we would feel pressure to adopt JavaScript on the back end, especially now that there are very good server side options such as node and deno. Why maintain two separate code bases? Why not reuse domain logic on the client-side as well as the server-side? When you choose a JavaScript heavy front end there are many forces pushing you to adopt the same langauge on the backend.

By using hypermedia, in contrast, you have more freedom in picking the back end technology appropriate for the problem domain you are addressing. You certainly aren't writing your server side logic in HTML, and every major programming langauge has at least one good templating library that can produce HTML cleanly.

If we are doing something in big data, perhaps we pick Python, which has tremendous support for that domain. If we are doing AI, perhaps we pick Lisp, leaning on a language with a long history in that area of research. Perhaps we prefer functional programming and wish to use OCaml or Haskell. Maybe you just really like Julia. Again, by using hypermedia as our front end technology, we are freed up to make any of these choices because there isn't a large JavaScript front end code base pressuring us to adopt JavaScript on the back end.

In the htmx community, we call this the HOWL stack: Hypermedia On Whatever you'd Like. We like the idea of a multi-language future. To be frank, a future of total JavaScript dominance (with maybe some TypeScript throw in) sounds pretty boring to us. We'd prefer to see many different language communities, each with their own strengths and cultures, participating in the web development world via the power of hypermedia. HOWL.

1.2. Contact.App Functionality

OK, let's get down to brass tacks: what will Contact.app do? Initially, it will provide the following functionality:

- Provide a list of contacts, including first name, last name, phone and email address
- Provide the ability to search the list of contacts
- Provide the ability to add a new contact to the list
- Provide the ability to view the details of a contact on the list
- Provide the ability to edit the details of a contact on the list
- Provide the ability to delete a contact from the list

So, you can see, this is a pretty basic CRUD application, the sort of thing that is perfect for an online web application.

1.3. Our Flask App

Flask is a very simple but flexible web framework for Python. This book is not a Flask book and we will not go into too much detail about it, but it is necessary to use **something** to produce our hypermedia, and Flask is simple enough that most web developers shouldn't have a problem following along. Let's go over the basics.

A Flask application consists of a series of *routes* tied to functions that execute when an HTTP request to that path is made. Let's look at the first route in Contacts.app

```
@app.route("/")
def index():
    return redirect("/contacts")
```

Don't worry about the `@app` stuff, just note the first line is saying: "When someone navigates to the root of this web application, invoke the index() method"

This is followed by a simple function definition, `index`, which simply issues an HTTP Redirect to the path `/contacts`.

So when someone navigates to the root directory of our web application, we redirect them to the `/contacts` URL. Pretty simple and I hope nothing too surprising for you, regardless of what web framework or language you are used to.

1.3.1. Showing A Searchable List Of Contacts

Next let's look at the `/contacts` route:

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

Once again, we map a path, `/contacts` to a handling function, `contacts()`

The implementation here is a bit more elaborate: we check to see if a search query named `q` is part of the request (e.g. `/contacts/q=joe`). If so, we delegate to a `Contact` model to do the search and return all matching contacts. If not, we simply get all contacts. We then render a template, `index.html` that displays the given contacts.

Note that we are not going to dig into the code in the `Contact` domain object. The implementation of the `Contact class is not relevant to hypermedia, beyond the API that it provides us. We will treat it as a *resource* and we will provide hypermedia representations of that resource to clients, in the form of HTML.

Next let's take a look at the `index.html` template:

```
{% extends 'layout.html' %}

{% block content %}

    <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>

    <table>
        <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Phone</th>
            <th>Email</th>
            <th></th>
        </tr>
        </thead>
        <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></td> <a
href="/contacts/{{ contact.id }}">View</a></td>
        {% endfor %}
        </tbody>
    </table>

    <p>
        <a href="/contacts/new">Add Contact</a>
    </p>

{% endblock %}
```

This Jinja2 template should be a fairly easy to understand for anyone who has done web development:

- We extend a base template `layout.html` which provides the layout for the page (sometimes called "the chrome"): it imports any necessary CSS, and scripts, includes the `<head>` element, and so forth.
- We then have a simple form that allows you to search contacts by issuing a `GET` request to

`/contacts`. Note that the input in this form keeps its value set to the value that is submitted with the name `q`.

- We then have a simple table as has been used since time immemorial on the web, where we iterate over all the `contacts` and display a row for each one
  - Recall that `contacts` has been either set to the result of a search or to all contacts, depending on what exactly was submitted to the server.
  - Each row has two anchors in it: one to edit and one to view the contact associated with that row
- Finally, we have an anchor tag that leads to a page that we can create new Contacts on

Note that in Jinja2 templates, we use `{{}}` to embed expression values (we use this to preserve the search value for example) and we use `{% %}` for directives, like iteration.

So far, so hypermedia! Notice that this template provides all the functionality necessary to both see all the contacts, search them and create a new one. It does this without the browser knowing a thing about Contacts or anything else: it just knows how to recieve and render HTML. This is a truly REST-ful application!

1.3.2. Adding A New Contact

To add a new contact, a user clicks on the "Add Contact" link above. This will issue a `GET` request to the `/contacts/new` URL, which is handled by this bit of code:

```
@app.route("/contacts/new", methods=['GET'])
def contacts_new_get():
    return render_template("new.html", contact=Contact())
```

Here we simply render a `new.html` template with, well, a new Contact. (`Contact()` is the python syntax for creating a new instance of the `Contact` class.)

Let's look at the `new.html` Jinja2 template:

```
{% extends 'layout.html' %}

{% block content %}

<form action="/contacts/new" method="post">
    <fieldset>
        <legend>Contact Values</legend>
        <div class="table rows">
            <p>
                <label for="email">Email</label>
                <input name="email" id="email" type="text" placeholder="Email"
value="{{ contact.email or '' }}">
                <span class="error">{{ contact.errors['email'] }}</span>
            </p>
            <p>
                <label for="first_name">First Name</label>
                <input name="first_name" id="first_name" type="text"
placeholder="First Name" value="{{ contact.first or '' }}">
                <span class="error">{{ contact.errors['first'] }}</span>
            </p>
            <p>
                <label for="last_name">Last Name</label>
                <input name="last_name" id="last_name" type="text" placeholder="Last
Name" value="{{ contact.last or '' }}">
                <span class="error">{{ contact.errors['last'] }}</span>
            </p>
            <p>
                <label for="phone">Phone</label>
                <input name="phone" id="phone" type="text" placeholder="Phone"
value="{{ contact.phone or '' }}">
                <span class="error">{{ contact.errors['phone'] }}</span>
            </p>
        </div>
        <button>Save</button>
    </fieldset>
</form>

<p>
    <a href="/contacts">Back</a>
</p>

{% endblock %}
```

Here you can see we render a simple form which issues a `POST` to the `/contacts/new` path and, thus should be handled by our logic above.

The form has a set of fields corresponding to the Contact and is populated with the values of the contact that is passed in.

Note that each form input also has a `span` element below it that displays an error message

associated with the field, if any.

It is worth pointing out something that is easy to miss: here we are again seeing the flexibility of hypermedia! If we add a new field, or change the logic around how fields are validated or work with one another, this new state of affairs is simply reflected in the hypermedia response given to users. A users will see the update content and be able to work with it. No software update required!

So, now we need to handle the `POST` that this form makes to create a new Contact.

To do so, we add another route that uses the same path but handles the `POST` method instead of the `GET`:

```
@app.route("/contacts/new", methods=['POST'])
def contacts_new():
    c = Contact(None, request.form['first_name'], request.form['last_name'],
request.form['phone'],
                request.form['email'])
    if c.save():
        flash("Created New Contact!")
        return redirect("/contacts")
    else:
        return render_template("new.html", contact=c)
```

Here we see a bit more complicated logic that we have seen in our other handlers, but not by very much:

- We create a new Contact, again using the `Contact()` syntax in python to construct the object. We pass in the values submitted by the user in the form by using the `request.form` object in Flask. This is a simple helper that allows us to access form values in a familiar HashMap-like manner.
- If we are able to save the contact (that is, there were no validation errors), we create a *flash* message indicating success and redirect the browser back to the list page. A flash is a common feature in web frameworks that allows you to store a message that will be available on the *next* request, typically in a cookie or in a session store.
- If we are unable to save the contact, we rerender the `new.html` template with the contact so it can provide feedback to the user as to what validation failed.

Note that, in the case of a successful creation of a contact, we have implemented the Post/Redirect/Get pattern we discussed earlier.

This is about as complicated as our application will get, even when we look at adding more advanced htmx-based behavior and this simplicity is, again, a great selling point of the hypermedia approach!

1.3.3. Viewing The Details Of A Contact

To view the details of a Contact, a user will click on the "View" link on one of the rows in thel ist of contacts.

This will take them to the path `/contact/<contact id>` (e.g. `/contacts/22`). Note that this is a common pattern in web development: Contacts are being treated as resources and are organized in a coherent manner:

- If you wish to view all contacts, you issue a `GET` to `/contacts`
- If you wish to get a hypermedia representation allowing you to create a new contact, you issue a `GET` to `/contacts/new`
- If you wish to view a specific contacts (with, say, and id of `42), you issue a `GET` to `/contacts/42`

It is easy to quibble about what particular path scheme you should use ("Should we `POST` to `/contacts/new` or to `contacts`) but what is more important is the overarching idea of resources and the hypermedia representations of them.

Here is what the controller logic looks like:

```
@app.route("/contacts/<contact_id>")
def contacts_view(contact_id=0):
    contact = Contact.find(contact_id)
    return render_template("show.html", contact=contact)
```

Very simple, we just look the Contact up by id, which is extracted from the end of the path automatically by Flask, based on the route mapping, and display the contact with the `show.html` template.

The `show.html` template looks like this:

```
{% extends 'layout.html' %}

{% block content %}

<h1>{{contact.first}} {{contact.last}}</h1>

<div>
<div>Phone: {{contact.phone}}</div>
<div>Email: {{contact.email}}</div>
</div>

<p>
<a href="/contacts/{{contact.id}}/edit">Edit</a>
<a href="/contacts">Back</a>
</p>


{% endblock %}
```

Another very simple template that just displays the information about the contact in a nice format, and includes links to edit the contact as well as to go back to the list of contacts.

1.3.4. Editing The Details Of A Contact

Editing a contact is definitely more interesting than viewing one.

Here is the Flask code to get the edit view for a contact:

```
@app.route("/contacts/<contact_id>/edit", methods=["GET"])
def contacts_edit_get(contact_id=0):
    contact = Contact.find(contact_id)
    return render_template("edit.html", contact=contact)
```

So, again we look the contact up, but this time we render the `edit.html` template instead, which looks like this:

```
{% extends 'layout.html' %}

{% block content %}

    <form action="/contacts/{{ contact.id }}/edit" method="post">
        <fieldset>
            <legend>Contact Values</legend>
            <div class="table rows">
                <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>
                </p>
                <p>
                    <label for="first_name">First Name</label>
                    <input name="first_name" id="first_name" type="text"
placeholder="First Name"
                           value="{{ contact.first }}">
                    <span class="error">{{ contact.errors['first'] }}</span>
                </p>
                <p>
                    <label for="last_name">Last Name</label>
                    <input name="last_name" id="last_name" type="text"
placeholder="Last Name"
                           value="{{ contact.last }}">
                    <span class="error">{{ contact.errors['last'] }}</span>
                </p>
                <p>
                    <label for="phone">Phone</label>
                    <input name="phone" id="phone" type="text" placeholder="Phone"
value="{{ contact.phone }}">
                    <span class="error">{{ contact.errors['phone'] }}</span>
                </p>
            </div>
            <button>Save</button>
        </fieldset>
    </form>

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

    <p>
        <a href="/contacts/">Back</a>
    </p>

{% endblock %}
```

This looks very similar to the `new.html` template. In fact, if we were to factor (that is, organize or

split up) this application properly, we would probably share the form between the two views to avoid redundancy and only have to maintain the form in one place.

Since we are keeping the application simple, for now we will keep them separate.

Factoring Your Applications

One thing that often trips people up who are coming to hypermedia applications from a JavaScript background is the notion of "components". In JavaScript-oriented applications it is common to break your app up into small client-side components that are then composed together. These components are often developed and tested in isolation and provide a nice abstraction for developers to create testable code.

In hypermedia applications, in contrast, you factor your application on the server side. As we said, the above form could be refactored into a shared template between the edit and create templates, allowing you to achieve a reusable and DRY (Don't Repeat Yourself) implementation.

Note that factoring on the server side tends to be coarser-grained than on the client side: you tend to split out common *sections* rather than create lots of individual components. This has both benefits (it tends to be simple) as well as drawbacks (it is not nearly as isolated as client-side components) .

Overall, however, a properly factored server-side hypermedia application can be extremely DRY!

Returning to the `edit.html` template, we again see a form that issues a `POST` request, now to the edit URL for a given contact. The fields are populated by the contact that is passed in from the control logic.

Below the main editing form, we see a second form that allows you to delete a contact. It does this by issuing a `POST` to the `/contacts/<contact id>/delete` path. Note that we aren't we issuing a `PUT` or `DELETE` HTTP request here because unfortunately those HTTP request types are not available. (Sure would be nice if they were!)

Finally, there is a simple hyperlink back to the list of contacts.

Here is the Flask route that handles the `POST` from the edit form:

```
@app.route("/contacts/<contact_id>/edit", methods=["POST"])
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():
        flash("Updated Contact!")
        return redirect("/contacts/" + str(contact_id))
    else:
        return render_template("edit.html", contact=c)
```

This logic is very similar to the logic for adding a new contact. The only real difference is that, rather than creating a new Contact, we look up a contact by id and then call `update()` on it with the values that were entered in the form.

This consistency between our CRUD operations is one of the nice simplifying aspects of traditional CRUD web applications!

1.3.5. Deleting A Contact

The delete functionally of our application only involves a bit of Flask code which is invoked when a `POST` request is made to the `/contacts/<contact id>/delete` path:

```
@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")
```

Here we simply look up and delete the contact in question and redirect back to the list of contacts.

There is no need for a template in this case, the hypermedia response is simply a redirect back to the list of contacts, along with a flash message notifying the user that the contact has been deleted.

1.3.6. Summary

So that's our simple contact application. Hopefully the Flask and Jinja2 code is simple enough that you were able to follow along easily, even if Python isn't your preferred language or Flask isn't your preferred web application framework.

Now, admittedly, this isn't a huge, sophisticated application at this point, but it demonstrates many of the aspects of traditional, web 1.0 applications: CRUD, the Post/Redirect/Get pattern, working with domain logic in a controller, organizing our URLs in a coherent, resource-oriented manner.

And, furthermore, this is a *deeply RESTful* web application. Without thinking about it very much we have been using HATEOAS to perfection. I would be that this simple little app we have built is more REST-ful than 99% of all JSON APIs ever built, and it was all effortless: just by virtue of using a

hypermedia, HTML, we naturally fall into the REST-ful network architecture.

Great, so what's the matter with this little web app? Why not end here and go off to develop the old web 1.0 style applications we used to build? Well, at some level, nothing is wrong with it. Particularly for an application of this size and complexity, this older way of building web apps is likely fine. However, there is that clunkiness we mentioned earlier when discussing older web applications: every request replaces the entire screen and there is often a noticeable flicker when navigating between pages. You lose your scroll state. You have to click things a bit more than you might in a more sophisticated application. It just doesn't have the same feel as a "modern" web application, does it?

So, are we going to have to adopt JavaScript after all? Pitch hypermedia in the bin, install NPM and start pulling down thousands of JavaScript dependencies, in the name of a better user experience? Well, I wouldn't be writing this book if that were the case.

It turns out you can improve the user experience of this application *without* abandoning the hypermedia architecture. This can be accomplished with htmx, a small JavaScript library that eXtends HTML (hence, htmx) in a natural manner. In the next few chapters we will take a look at this library and how it can be used to build surprisingly interactive user experiences, all within the origina, REST-ful architecture of the web.