Hypermedia In Action

1. A Simple Web Application

This chapter covers:

- Picking a "web stack" to build our sample hypermedia application in
- A brief introduction to Flask & Jinja2 for Server Side Rendering (SSR)
- An overview of the functionality of our sample hypermedia application, Contact.App
- Implementing the basic CRUD (Create, Read, Update, Delete) operations + search for Contact.App

1.1. A Simple 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!) It will also be easy to incrementally improve the application by taking advantage of hypermedia-oriented technologies like htmx.

By the time we are finished with the application it will have some very slick features that most developers today would assume requires the use of sophisticated client-side infrastructure. We will, instead, implement these features entirely using the hypermedia approach, but enhanced with htmx and other libraries that stay within this paradigm.

1.1.1. Picking A "Web Stack" To Use

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 Side" or "Web" stack, and there are literally hundreds of options to choose from, all with passionate followers. You probably have a web framework that you prefer and, while I wish we could write this book for every possible stack out there, in the interest of berevity we can only pick one. For this book we are going to use the following stack: Python as our programming language, Flask as our web framework, and Jinja2 for our server-side templating language.

Why pick this particular stack? I am not a day-to-day Python programmer, so it's not an obvious choice for me! But this particular stack has a number of advantages.

First off, python is the most popular programming language today by most industry measures. More importantly, even if you don't know or like Python, it is very easy to read. As a veteran of the programming language wars of the 90's and early aughts, I understand how passionate people are around programming languages, and I hope python is a "least worst" choice for readers who are not pythonistas.

Flask was picked because it is very light weight and does not impose a lot of structure on top of the basics of HTTP routing. This bare bones approach isn't for everyone: in the python community, for

example, many people prefer the "Batteries Included" nature of Django, for example, where lots of functionality is available out of the box.

I understand that, but for demonstration purposes, I feel that an unopionated and light-weight library will make it easier for non-Python developers to follow along by minimizing the amount of code required on the server side. Anyone who prefers django or some other Python web framework, or some other language entirely for that matter, should be able to easily convert the Flask examples into their native framework.

Jinja2 templates were picked because they are the default templating language for Flask. They are simple enough and standard enough that most people who understand any server side (or client side) templating library will be able to understand them reasonably 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 (Hypermedia On Whatever you'd Like) Stack

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! You just need to be able to produce HTML with it.

Consider if we were instead building a web application with a large JavaScript-based SPA front end. We would almost certainly feel pressure to adopt JavaScript on the back end. We already would have a ton of code written in JavaScript. Why maintain two separate code bases? Why not reuse domain logic on the client-side as well as the server-side? There are now that very good server side options for writing JavaScript code like node and deno. Why not just a single language for everything?

So, as you may have felt yourself, if you choose a JavaScript heavy front end there are many forces pushing you to adopt JavaScript on the back end.

In contrast, by using a hypermedia-based front end you have a lot more freedom in picking the back end technology appropriate to the problem domain you are addressing. You certainly aren't writing your server side logic in HTML! And every major programming language has at least one good templating library that can produce HTML cleanly, often more.

If we are doing something in big data, perhaps you'd like to use Python, which has tremendous support for that domain. If we are doing AI, perhaps you'd like to use Lisp, leaning on a language with a long history in that area of research. Maybe you are a functional programming enthusiast and want to use OCaml or Haskell. Maybe you just really like Julia or Nim. All perfectly valid reasons for choosing a particular server side technology! By using hypermedia as your *front end* technology, you are freed up to adopt any of these choices. There simply isn't a large JavaScript front end code base pressuring you 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, multi-framework future in web development. 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 and web framework communities, each with their own strengths and cultures, participating in the web development world, all through the power of hypermedia.

That sounds better to us, and that's HOWL.

1.2. A Brief Introduction to Flask & Our First Route

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, as we said, it is necessary to use *something* to produce our hypermedia on the server side, 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 a given path is made.

Let's look at the first "route" definition in our application. It will be a simple redirect, such that when a user goes to the root of our web application, `/`, they will be redirected to the `/contacts` path instead. Redirection is an HTTP feature where, when a user requests one URL, they are sent to another on, and is a basic piece of web functionality that is well supported in most web frameworks.

Let's create our first route definition. In the following python code you will see the `@app` symbol. This refers to the flask application object. Don't worry too much about how it has been set up, just understand that it is an object that encapsulates the mapping of requests to some path to some python logic to be executed on the server when a request to that path is made.

Here is the code:

```
@app.route("/") ①
def index(): ②
    return redirect("/contacts") ③
```

① Establishes we are mapping the `/` path as a route

② The next method is the handler for that route

③ Redirect the request to a new path

In this case, we wish to say "When someone navigates to the root of this web application, redirect to /contacts". The Flask pattern for doing this is to use the `route()` method on the Flask application object, and pass in the path you wish this route to handle. In this case we pass in the root or `/` path, as a string, to the `@app.route()` method. This establishes a path that Flask will handle.

This route declaration is then followed by a simple function definition, `index()`. The Flask approach for defining logic to handle requests to a given route is that it will take the *next* function defined after the route has been declared and make function that the handler for that route. (Note that the name of the function doesn't matter, we can call it whatever we'd like. In this case I chose `index()` because that fits with the route we are handling: the root "index" of the web applications.) So we have the `index()` function immediately following our route definition for the root, and this will become the handler for the root URL in our web application.

The body of the `index()` function simply returns the result of calling a `redirect()` function with the path we with to redirect to, in this case `/contacts`, passedin as a string. This simple handler implementation will trigger an HTTP Redirect to that path, achieving what we desire for this route.

So, in summary, given the functionality above, when someone navigates to the root directory of our web application, Flask will redirect them to the `/contacts` path. Pretty simple, and I hope nothing too surprising for you, regardless of what web framework or language you are used to!

1.3. Contact.App Functionality

OK, with that brief introduction to Flask out of the way, let's get down to specifying and implementing our application. 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, as you can see, this is a pretty basic CRUD application, the sort of thing that is perfect for an old-school web 1.0 application.

1.3.1. Showing A Searchable List Of Contacts

Let's look at our first "real" bit of functionality: the ability show all the contacts in our system in a list (really, in a table).

This functionality is going to be found at the `/contacts` path, which is the path our previous route is redirecting to.

We will use the `@app` flask instance to route the `/contacts` path and then define a handler function, `contacts()`. This function is going to do one of two things:

- If there is a search term, it filter all contacts matching that term
- If not, it will simply return all contacts in our database.

Here is the code:

```
@app.route("/contacts")
def contacts():
    search = request.args.get("q") ①
    if search:
        contacts_set = Contact.search(search) ②
    else:
        contacts_set = Contact.all() ③
    return render_template("index.html", contacts=contacts_set) ④
```

① Look for the query parameter named `q`, which stands for "query"

② If the parameter exists, call the `Contact.search()` function with it

③ If not, call the `Contact.all()` function

④ pass the result to the `index.html` template to render to the client

We see the usual routing code we saw in our first example, but then we see some more elaborate code in the handler function. First, we check to see if a search query parameter named `q` is part of the request. The "query string" is part of the URL specification and you are probably familiar with it. Here is an example URL with a query string in it: `https://example.com/contacts?q=joe`. The query string is everything after the `?` and is a name-value pair format. In this case, the query parameter `q` is set to the string value `joe`.

To get back to the code, if a query parameter is found, we call out to the `search()` method on the `Contact` model to do the actual search and return all matching contacts. If the query parameter is *not* found, we simply get all contacts by invoking the `all()` method on the `Contact` object.

Finally, we then render a template, `index.html` that displays the given contacts, passing in the results of whichever function we ended up calling.

Note that we are not going to dig into the code in the `Contact` class. The implementation of the `Contact` class is not relevant to hypermedia, we will ask you to simply accept that it is a "normal" domain model class, and the methods on it act in the "normal" manner. We will treat `Contact` as a *resource* and will provide hypermedia representations of that resource to clients, in the form of HTML generated via server side templates.

The List & Search Template

Now we need to take a look at the template that we are going to render in our response to the client. In this HTML response we want to have a few things:

- A list of any matching or all contacts
- A search box that a user may type a value into and submit for searches
- A bit of surrounding "chrome": a header and footer for the website that will be the same regardless of the page you are on

Recall we are using the Jinja2 templating language here. In Jinja2 templates, we use `{{}}` to embed expression values and we use `{% %}` for directives, like iteration or including other content. Jinja2 is very similar to other templating languages and I hope you are able to follow along easily.

Let's look at the first few lines of code in our `index.html` template:

```
{% extends 'layout.html' %} ①

{% block content %} ②

    <form action="/contacts" method="get" class="tool-bar">  ③
            <label for="search">Search Term</label>
            <input id="search" type="search" name="q" value="{{ request.args.get('q')
or '' }}"/> ④
            <input type="submit" value="Search"/>
     </form>
```

① Set the layout template for this template

② Delimit the content to be inserted into the layout

③ Create a search form that will issue an HTTP `GET` to the `/contacts` page

④ Create an input that a query can be typed into to search contacts

The first line of code references a base template, `layout.html`, with the `extends` directive. This layout template provides the layout for the page (again, sometimes called "the chrome"): it imports any necessary CSS and scripts, includes a `<head>` element, and so forth.

The next line of code declares the `content` section of this template, which is the content that will be included within the "chrome" of the layout template.

Next we see our first true bit of HTML: a simple form that allows you to search contacts by issuing a `GET` request to `/contacts`. Note that the value of this input is set to the expression `{{ request.args.get('q') or '' }}`. This expression is evaluated by Jinja templates and inserted as escaped text into the input. What this is doing is preserving the query value between requests, so if you search for "joe" then this input will have the value "joe" in it when the page re-renders.

The next bit of Jinja template has the actual contacts table code in it:

```
    <table>
        <thead>
        <tr>
            <th>First</th> <th>Last</th> <th>Phone</th> <th>Email</th> <th></th>①

        </tr>
        </thead>
        <tbody>
        {% for contact in contacts %} ②
            <tr>
                <td>{{ contact.first }}</td>
                <td>{{ contact.last }}</td>
                <td>{{ contact.phone }}</td>
                <td>{{ contact.email }}</td> ③
                <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
                    <a href="/contacts/{{ contact.id }}">View</a></td> ④
            </tr>
        {% endfor %}
        </tbody>
    </table>
```

① We output some headers for our table

② We iterate over the contacts that were passed in to the template

③ We output the values of the current contact, first name, last name, etc. in columns

④ An "operations" column that has links embedded in it to edit or view the contact details

Here we are into the "meat" of the page: we construct a table with appropriate headers matching the data we are going to show for each contact. We iterate over the contacts that were passed into the template by the handler method using the `for` loop directive in Jinja2. We then construct a series of rows, one for each contact, where we render the first and last name, phone and email of the contact as table cells in the row.

Finally, we have an additional cell that includes two links:

- A link to the "Edit" page for the contact, located at `/contacts/{{ contact.id }}/edit` (e.g. For the contact with id 42, the edit link will point to `/contacts/42/edit`)
- A like to the "View" page for the contact `/contacts/{{ contact.id }}` (using our previous contact

example, the show page would be at `/contacts/42`)

This is our contacts table.

Finally, we have a bit of end-matter: a link to add a new contact and a directive to close up the `content` block:

```
    <p>
        <a href="/contacts/new">Add Contact</a> ①
    </p>

{% endblock %} ②
```

① A link to the page that allows you to create a new contact

② The closing element of the `content` block

And that's our template! Using this server side template, in combination with our handler method, we can respond with an HTML *representation* of all the contacts requested. So far, so hypermedia! Notice that our template, when rendered, provides all the functionality necessary to see all the contacts and search them, and also provides links to edit them, view details of them or even create a new one. And it does all this without the browser knowing a thing about Contacts! The browser just knows how to issue HTTP requests and render HTML. This is a truly REST-ful application!

1.3.2. Adding A New Contact

The next bit of functionality that we will add to our application is the ability to add new contacts. To do so, we are going to need to handle that `/contacts/new` URL referenced in the "Add Contact" link above. Note that when a user clicks on that link, the browser will issue a `GET` request to the `/contacts/new` URL. The other routes we have been looking at were using `GET` as well, but we are actually going to use two different HTTP methods for this bit of functionality: an HTTP `GET` and an HTTP `POST`, so we are going to be explicit when we declare this route.

Here is our code:

```
@app.route("/contacts/new", methods=['GET']) ①
def contacts_new_get():
    return render_template("new.html", contact=Contact()) ②
```

① We declare a route, explicitly handling `GET` requests to this path

② We render the `new.html` template, passing in a new contact object

Pretty simple! We just render a `new.html` template with, well, a new Contact, as you might expect! (Note that `Contact()` is the python syntax for creating a new instance of the `Contact` class.)

So the handler code for this route is very simple. The `new.html` Jinja2 template, in fact, is more complex. For the remaining templates I am not going to include the starting layout directive or the content block declaration, but you can assume they are the same unless I say otherwise. This will let us focus on the "meat" of the template.

If you are familiar with HTML you are probably expecting a form element here, and you will not be disappointed. We are going to use the standard form element for collecting contact information and submitting it to the server.

```
<form action="/contacts/new" method="post"> ①
    <fieldset>
        <legend>Contact Values</legend>
        <p>
            <label for="email">Email</label> ②
            <input name="email" id="email" type="email" placeholder="Email" value="{{
contact.email or '' }}"> ③
            <span class="error">{{ contact.errors['email'] }}</span> ④
        </p>
```

① A form that will submit to the `/contacts/new` path, using an HTTP `POST` request

② A label for the first form input

③ the first form input, of type email

④ Any error messages associated with this field

In the first line of code we create a form that will submit back *to the same path* that we are handling: `/contacts/new`. Rather than issuing an HTTP `GET` to this path, however, we will issue an HTTP `POST` to it. This is the standard way of signalling via HTTP that you wish to create a new resource, rather than simply get a representation of it.

We then have a label and input (always a good practice) that capture the email of the new contact in question. The "name" of the input is "email" and, when this form is submitted, the value of this input will be submitted in the `POST` request, associated with the "email" key.

Next we have inputs for the other fields for contacts:

```
        <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>
```

Finally, we have a button that will submit the form, the end of the form tag, and a link back to the main contacts table:

```
        <button>Save</button>
    </fieldset>
</form>

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

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, remove a 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 representation given to users. A user will see the updated new content and be able to work with it, no software update required!

Handling The Post

The next step in our application is to handle the `POST` that this form makes to `/contacts/new` to create a new Contact.

To do so, we need to add another route that uses the same path but handles the `POST` method instead of the `GET`. We will take the submitted form values and attempt to create a Contact. If it works, we will redirect to the list of contacts and show a success message. If it doesn't then we will show the new contact form again, rendering any errors that occurred in the HTML so the user can correct them.

Here is our controller code:

```
@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']) ①
    if c.save(): ②
        flash("Created New Contact!")
        return redirect("/contacts") ③
    else:
        return render_template("new.html", contact=c) ④
```

① We construct a new contact object with the values from the form

② We try to save it

③ If it succeeds we "flash" a success message and redirect back to the `/contacts` page

④ If not, we rerender the form, showing any errors to the user

The logic here is a bit more complex than other handler methods we have seen, but not by a whole lot. The first thing we do is 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, provided by flash Flask. This object allows us to access form values in a convenient and easy to read syntax. Note that we pick out each value based on the `name` associated with each input in the form.

We also pass in `None` as the first value to the `Contact` constructor. This is the "id" parameter, and by passing in `None` we are signaling that it is a new contact, and needs to have an ID generated for it.

Next, we call the `save()` method on the Contact object. This returns `true` if the save is successful, and `false` if the save is unsuccessful, for example if one of the fields has a bad value in it. (Again, we are not going to dig into the details of how this model object is implemented, our only concern is using it to generate hypermedia responses.)

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.

Finally, if we are unable to save the contact, we rerender the `new.html` template with the contact. This will show the same template as above, but the inputs will be filled in with the submitted values, and any errors associated with the fields will be rendered to 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.

Believe it or not, this is about as complicated as our handler logic will get, even when we look at adding more advanced htmx-based behavior. Simplicity is a great selling point of the hypermedia approach!

1.3.3. Viewing The Details Of A Contact

The next piece of functionality we will implement is the details page for a Contact. The user will navigate to this page by clicking the "View" link in one of the rows in the list of contacts. This will take them to the path `/contact/<contact id>` (e.g. `/contacts/42`). Note that this is a common pattern in web development: Contacts are being treated as resources and the URLs around these resources 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`?") and we have seen *lots* of arguments about one approach versus another. What we feel is more important is the overarching idea of resources and the hypermedia representations of them: just pick a schema you like and stay consistent.

Our handler logic for this route is going to be *very* simple: we just look the Contact up by id, embedded in the path of the URL for the route. To extract this ID we are going to need to introduce a final bit of Flack functionality: the ability to call out pieces of a path and have them automatically extracted and then passed in to a handler function.

Let's look at the code

```
@app.route("/contacts/<contact_id>") ①
def contacts_view(contact_id=0): ②
    contact = Contact.find(contact_id) ③
    return render_template("show.html", contact=contact) ④
```

① Map the path, with a path variable named `contact_id`

② The handler takes the value of this path parameters

③ Look up the corresponding contact

④ Render the `show.html` template

You can see the syntax for extracting values from the path in the first line of code, you enclose the part of the path you wish to extract in `<>` and give it a name. This component of the path will be extracted and then passed into the handler function, via the parameter with the same name. So, if you were to navigate to the path `/contacts/42` then the value `42` would be passed into the `contacts_view()` function for the value of `contact_id`.

Once we have the id of the contact we want to look up, we load it up using the `find` method on the `Contact` object. We then pass this contact into the `show.html` template and render a response.

1.3.4. Viewing The Details Of A Contact

Our `show.html` template is relatively simple, just showing the same information as the table but in a

slightly different format (perhaps for printing.) If we add functionality like "notes" to the application later on, however, this will give us a good place to show them.

Again, I will omit the "chrome" and focus on the meat of the template:

```
<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>
```

We simply render a nice First Name abd Last Name header with the additional contact information as well as a link to edit it or to navigate back to the list of contacts. Simple but effective hypermedia!

1.3.5. Editing And Deleting A Contact

Editing a contact is going to look very similar to creating a new contact. As with adding a new contact, we are going to need two routes that handle the same path, but using different HTTP methods: a `GET` to `/contacts/<contact_id>/edit` will return a form allowing you to edit the contact with that ID and the `POST` will update it.

We will also piggyback the ability to delete a contact along with this editing functionality. To do this we will need to handle a `POST` to `/contacts/<contact_id>/delete`.

Let's look at the code to handle the `GET`, which, again, will return an HTML representation of an editing interface for the given resource:

```
@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)
```

As you can see this looks an awful lot like our "Show Contact" functionality. In fact, it is nearly identical except for the template that we render: here we render `edit.html` rather than `show.html`! There's that simplicity we talked about again!

While our handler code looked similar to the "Show Contact" functionality, our template is going to look very similar to the template for the "New Contact" functionality: we are going to have a form that submits values to the same URL used to `GET` the form (see what I did there?) and that presents all the fields of a contact as inputs, along with any error messages (we will even reuse the same Post-Redirect-Get trick!)

Here is the first bit of the form:

```
    <form action="/contacts/{{ contact.id }}/edit" method="post"> ①
        <fieldset>
            <legend>Contact Values</legend>
              <p>
                  <label for="email">Email</label>
                  <input name="email" id="email" type="text" placeholder="Email"
value="{{ contact.email }}"> ②
                  <span class="error">{{ contact.errors['email'] }}</span>
              </p>
```

① We issue a `POST` to the `/contacts/{{ contact.id }}/edit` path

② As with the `new.html` page, we have an input tied to the contact's properties

Nearly identical to our `new.html` form, except that this form is going to submit a `POST` to a different path, based on the id of the contact that is passed in.

Following this we have the remainder of our form, again very similar to the `new.html` template, and our submit button to submit the form.

```
              <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>
            <button>Save</button>
        </fieldset>
    </form>
```

In the final part of our template we have a small difference between the `new.html` and `edit.html`. Below the main editing form, we include a second form that allows you to delete a contact. It does this by issuing a `POST` to the `/contacts/<contact id>/delete` path. Sure would be nice if we could

issue a `DELETE` request instead, but unfortunately that isn't possible in plain HTML!

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

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

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

Given all the similarities between the `new.html` and `edit.html` templates, you may be wondering why we are not *refactoring* these two templates to share logic between them. That's a great observation and, in a production system, we would probably do just that. For our purposes, however, since the app is so small and simple, we will leave the templates 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!

Handling The Post

Next we need to handle the HTTP `POST` request that the form in our `edit.html` template submits. We will declare another route that handles the path as the `GET` above.

Here is the definition:

```
@app.route("/contacts/<contact_id>/edit", methods=["POST"]) ①
def contacts_edit_post(contact_id=0):
    c = Contact.find(contact_id) ②
    c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email']) ③
    if c.save(): ④
        flash("Updated Contact!")
        return redirect("/contacts/" + str(contact_id)) ⑤
    else:
        return render_template("edit.html", contact=c) ⑥
```

① Handle a `POST` to `/contacts/<contact_id>/edit`

② Look the contact up by id

③ update the contact with the new information from the form

④ Attempt to save it

⑤ If successful, flash a success message and redirect to the show page for the contact

⑥ If not successful, rerender the edit template, showing any errors.

The logic in this handler is very similar to the logic in the handler 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 the `update()` method on it with the values that were entered in the form.

Once again, this consistency between our CRUD operations is one of the nice, simplifying aspects of traditional CRUD web applications!

1.3.6. Deleting A Contact

We piggybacked delete functionality into the same template used to edit a contact. That form will issue an HTTP `POST` to `/contacts/<contact_id>/delete` that we will need to handle and delete the contact in question.

Here is what the controller looks like

```
@app.route("/contacts/<contact_id>/delete", methods=["POST"]) ①
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete() ②
    flash("Deleted Contact!")
    return redirect("/contacts") ③
```

① Handle a `POST` the `/contacts/<contact_id>/delete` path

② Look up and then invoke the `delete()` method on the contact

③ Flash a success message and redirect to the main list of contacts

The handler code is very simple since we don't need to do any validation or conditional logic: we simply look up the contact the same way we have been doing in our other handlers and invoke the

`delete()` method on it, then redirect back to the list of contacts with a success flash message.

No need for a template in this case!

1.3.7. Contact.App… Implemented!

Believe it or not, that's our entire 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. Again, I don't expect you to be a Python or Flask expert (I'm certainly not!) and you shouldn't need more than a basic understanding of how they work for the remainder of the book.

Now, admittedly, this isn't a large or sophisticated application, but it does demonstrate 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 *hypermedia-based* web application. Without even thinking about it (or maybe even understanding it!) we have been using REST, HATEOAS and all the other hypermedia concepts. I would bet 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.

So that's great. But 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 people used to build?

Well, at some level, nothing is wrong with it. Particularly for an application that is as simple as this one it, the older way of building web apps may be a fine approach!

However, the application does suffer from that "clunkiness" that we mentioned earlier when discussing web 1.0 applications: every request replaces the entire screen, introducing a noticeable flicker when navigating between pages. You lose your scroll state. You have to click around a bit more than you might in a more sophisticated web application. Contact.App, at this point, just doesn't feel like a "modern" web application, does it?

Well. Are we going to have to adopt JavaScript after all? Should we pitch this hypermedia approach in the bin, install NPM and start pulling down thousands of JavaScript dependencies, and rebuild the application using a "modern" JavaScript library like React?

Well, I wouldn't be writing this book if that were the case, now would I?!

No, I wouldn't. It turns out that we can improve the user experience of this application *without* abandoning the hypermedia architecture. One way this can be accomplished is to introduce htmx, a small JavaScript library that eXtends HTML (hence, htmx), to our application. 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 original hypermedia architecture of the web.

1.4. Summary

- A Hypermedia Driven Application is an application that primarily relies on hypermedia exchanges for its network architecture

- Web 1.0 applications are naturally Hypermedia Driven Applications
- Flask is a simple Python library for connecting routes to server-side logic, or handlers
- Jinja2 is a simple Python template library
- Combining them to implementing a basic CRUD-style application for managing contacts, Contacts.app, is surprisingly simple.
- We will be looking at how to address the UX problems associated with Web 1.0 applications next