Hypermedia In Action

1. Reintroducing Hypermedia

This chapter covers

- A reintroduction to the core concepts of hypermedia
- Why you might choose hypermedia over other approaches
- How hypermedia can be used to build modern web applications

Hypermedia is a universal technology today, nearly as common as electricity. Billions of people use a hypermedia-based systems every day, mainly by interacting with the *HyperText Markup Language (HTML)* being exchanged via the *HyperText Transfer Protocol (HTTP)* by using a Web Browser connected to the World Wide Web. People use these systems to get their news, check in on friends, buy things online, play games, send emails and so forth: the variety and sheer number of online services is truly astonishing.

And yet, despite this ubiquity, hypermedia itself is a strangely under-explored concept, left mainly to specialists. Yes, you can find a lot of tutorials on how to author HTML, create links and forms, etc. But it is rare to see a discussion of HTML *as a hypermedia*. This is in sharp contrast with the early web development era, when concepts like *Representational State Transfer (REST)* and *Hypermedia As The Engine of Application State (HATEOAS)* were constantly discussed and debated among developers.

It is sad to say, but in some circles today HTML is viewed resentfully: it is considered an awkward, legacy markup language that must be used build user interfaces in what are primarily Javascript-based applications, simply because HTML happens to be there, in the browser.

This as a shame, and we hope that with this book we can convince you that the hypermedia architecture is not simply a piece of legacy technology that we have to begrudgingly deal with. Instead, we aim to show you that it is a tremendously innovative, simple and *flexible* way to build robust distributed systems. Not only that, but the hypermedia approach deserves a seat at the table when you, a web developer, are considering what the architecture of your next online software system will be.

1.1. So, What Is Hypermedia?

> The English prefix "hyper-" comes from the Greek prefix "ὑπερ-" and means "over" or "beyond"… It signifies the overcoming of the previous linear constraints of written text.
>
> — Wikipedia, https://en.wikipedia.org/wiki/Hypertext

Right. So what is hypermedia? Simply, it is a media, for example a text, that includes non-linear branching from one location in the media to another, via, for example, hyperlinks embedded that the media.

You are probably more familiar with the term *hypertext*, from whose Wikipedia page the above quote is taken. Hypertext is a sub-set of hypermedia and much of this book is going to discuss how to build modern web applications with HTML, the HyperText Markup Language.

However, even when you build applications using HTML, there are nearly always other types of media involved: images, videos and so forth. Because of this, we prefer the term *hypermedia* a more appropriate for discussing applications built in this manner. We will use the term hypermedia for most of this book in order to capture this more general concept.

1.1.1. HTML

> In the beginning was the hyperlink, and the hyperlink was with the web, and the hyperlink was the web. And it was good.
>
> — Rescuing REST From the API Winter, https://intercoolerjs.org/2016/01/18/rescuing-rest.html

In order to help us understand the more general concepts of hypermedia, it would be worthwhile to take a brief look at a concrete and familiar example of the technology: HTML.

HTML is the most widely used hypermedia in existence, and this book naturally assumes that the reader has a reasonable familiarity with it. You do not need to be an HTML or CSS expert to understand the code in this book, but the better you understand the core tags and concepts of HTML, the more you will get out of the book.

Now, let’s consider the two defining hypermedia elements in HTML: the anchor tag (which produces a hyperlink) and the form tag.

Here is a simple anchor tag:

Listing 1. 1. A Simple Hyperlink

```
<a href="https://www.manning.com/">
  Manning Books
</a>
```

In a typical browser, this tag would be interpreted to mean: "Show the text 'Manning Books' in manner indicating that it is clickable and, when the user clicks on that text, issue an HTTP GET to the url `https://www.manning.com/`. Take the resulting HTML content in the body of the response and use it to replace the entire screen in the browser as a new document."

This is the main mechanism we use to navigate around the web today, and it is a canonical example of a hypermedia link, or a hyperlink.

So far, so good. Now let’s consider a simple form tag:

Listing 1. 2. A Simple Form

```
<form action="/signup" method="post">
  <input type="text" name="email" placeholder="Enter Email To Sign Up..."/>
  <button>Sign Up</button>
</form>
```

This bit of HTML would be interpreted by the browser roughly as: "Show a text input and button to the user. When the user submits the form by clicking the button or hitting enter in the input, issue an HTTP POST request to the path '/signup' on the site that served the current page. Take the resulting HTML content in the response body and use it to replace the entire screen in the browser."

I am omitting a few details and complications here: you also have the option of issuing an HTTP `GET` with forms, the result may *redirect* you to another URL and so on, but this is the crux of the form tag.

Here is a visual representation of these two hypermedia interactions:

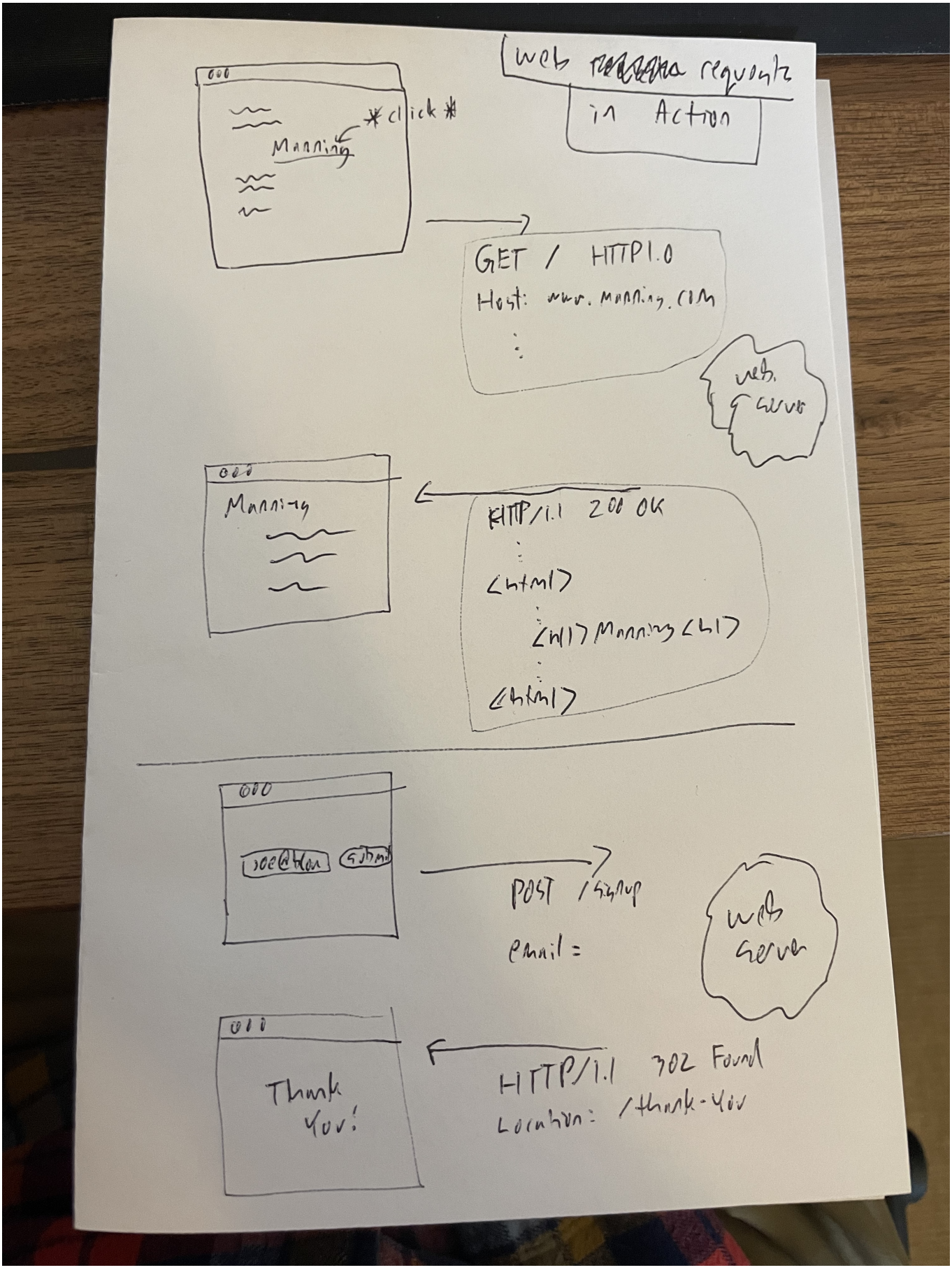

Figure 1. 1. HTTP Requests In Action

As someone interested in web development, the above diagram should look very familiar to you, perhaps even boring. But, despite its familiarity, consider the fact that the two above mechanisms are really the *only* easy ways to interact with a server in HTML. That's barely anything at all! And yet, armed with only these two tools, the early web was able to grow exponentially and offer a staggeringly large amount of online, dynamic functionality to an even more staggeringly large number of people!

This is strong evidence of the power of hypermedia. Even today, in a web development world increasingly dominated by large JavaScript-centric front end frameworks, many people choose to simply use vanilla HTML to achieve their goals and are often perfectly happy with the results.

So with just these two little tags, hypermedia manages to pack a heck of a punch!

1.1.2. So What Isn't Hypermedia?

So we've looked at the two ways to interact with a server via HTML. Now let's consider another approach to interacting with a server by issuing an HTTP request via JavaScript:

Listing 1. 3. Javascript

```
<button onclick="fetch('/api/v1/contacts') ❶
                 .then(response => response.json()) ❷
                 .then(data => updateTable(data))"> ❸
    Fetch Contacts
</button>
```

❶ Issue the request

❷ Convert the response to a JavaScript object

❸ Invoke the `updateTable()` function with the object

Here we have a button element in HTML that executes some JavaScript when it is clicked. That JavaScript will issue an HTTP `GET` request to `/api/v1/contacts` using the `fetch()` API, a popular API for issuing an "Asynchronous JavaScript and XML", or AJAX, requests. An AJAX request is like a normal HTTP request in many ways, but it is issued "behind the scenes" by the browser: the user does not see a request indicator like in normal links and forms, and it is up to the JavaScript code that issues the request to deal with the response.

Despite AJAX having XML as part of its acronym, today the HTTP response to this request would almost certainly be in the JavaScript Object Notation (JSON) format rather than XML. (That is a long story!)

The HTTP response to this request might look something like this:

Listing 1. 4. JSON

```
{ ❶
  "id": 42, ❷
  "email" : "json-example@example.org" ❸
}
```

❶ The start of a JSON object

❷ A property, in this case with the name `id` and the value `42`

❸ Another property, the email of the contact with this id

The JavaScript code above converts the JSON text received from the server into a JavaScript object, which is very easy when using the JSON notation. This object is then handed off to the `updateTable()` method. The `updateTable()` method would then update the UI based on the data that has been received from the server, perhaps appending this contact information to an existing table or replacing some other content with it. (We aren't going to show this code because it isn't important for our discussion.)

What is important to understand about this server interaction is that it is *not* using hypermedia. The JSON API being used here does not return a hypermedia-style response. There are no *hyperlinks* or other hypermedia-style controls in it. This is, rather, a *Data API*. It is returning simple, Plain Old JSON(POJ) formatted data. We say "POJ" here because, when XML was being used rather than JSON, the term for an API like this was "Plain Old XML", or POX. The term POX was disparaging at the time, sometimes called "The Swamp of POX", but, today, the POJ style of HTTP API is ubiquitous.

Now, because the response is in POJ and is *not* hypermedia, it is up to the code in the `updateTable()` method to understand how to turn this data into HTML. The code in `updateTable()` needs to know about the internal structure of this data, what the fields are named, how they relate to one another, how to update the data, and how to render this data to the browser. This last bit of functionality would typically be done via some sort of client-side templating library that generates HTML in memory in the browser based on data

passed into it.

Now, this bit of javascript, while very modest, is the beginnings of what has come to be called a Single Page Application (SPA): in this case, the application is no longer navigating between pages using hypermedia controls like anchor tags that interact with a server using hypermedia. Instead, the application is exchanging *plain data* with the server and updating the content within a single page, hence "Single Page Applications".

Today, of course, the vast majority of Simple Page Applications adopt far more sophisticated frameworks for managing their user interface than this simple example shows. Libraries like React, Angular, Vue.js, etc. are all popular ways to manage far more complex user interactions than our little demo. With these more complex frameworks you will typically work with a much more elaborate client-side model (that is, JavaScript objects stored locally in the browser's memory that represent the "model" or "domain" of your application.) You then update these JavaScript objects and allow the framework to "react" to those changes via infrastructure baked into the framework itself, which will have the effect of updating the user interface. (This is where the term "Reactive" programming comes from.)

At this point, if you adopt one of these popular libraries, you, the developer, rarely interact with hypermedia at all. You may it to build your user interface, but the anchor tag's natural behavior is de-emphasized and forms become a data collection mechanism. Neither interact with the server in their native language of HTML, and rather become user interface elements that drive local interactions with the in memory domain model, which is then synchronized with a server via JSON APIs.

So, admittedly, modern SPAs are much more complex than what we have going on in the above example. However, at the level of a *network architecture*, these more sophisticated frameworks are essentially equivalent to our simple example: they exchange Plain Old JSON with the server, rather than exchanging a hypermedia.

1.2. Why Use Hypermedia?

> The emerging norm for web development is to build a React single-page application, with server rendering. The two key elements of this architecture are something like:
>
> 1. The main UI is built & updated in JavaScript using React or something similar.
> 2. The backend is an API that that application makes requests against.
>
> This idea has really swept the internet. It started with a few major popular websites and has crept into corners like marketing sites and blogs.
>
> — Tom MacWright, https://macwright.com/2020/05/10/spa-fatigue.html

Tom is correct: JavaScript-based Single Page Applications have taken the web development world by storm, offering a far more interactive and immersive experience than the old, gronky, web 1.0 HTML-based application could. Some SPAs are even able to rival native applications in their user experience and sophistication.

So, why on earth would you abandon this new, increasingly standard (just do a job search for reactjs!) approach for an older and less discussed one like hypermedia?

Well, it turns out that, even in its original form, the hypermedia architecture has a number of advantages when compared with the JSON/Data API approach:

- It is an extremely simple approach to building web applications
- It survives network outages and changes relatively well
- It is extremely tolerant of content and API changes (in fact, it thrives on them!)

As someone interested in web development, these advantages should sound appealing to you. The first and last one, in particular, address two pain points in modern web development:

- Front end infrastructure has become extremely complex (sophisticated might be the nice way of saying it!)
- API churn is a huge pain for many applications

Taken together, these two problems have become known as "Javascript Fatigue": a general sense of exhaustion with all the hoops that are necessary to jump through to get anything done on the web.

And it’s true: the hypermedia architecture *can* help cure Javascript Fatigue. But you may

reasonably be wondering: so, if hypermedia is so great and can address these problems so obvious in the web development industry, why has it has been abandoned web developers today? After all, web developers are a pretty smart lot. Why wouldn't they use this obvious, native web technology?

There are two related reasons for this somewhat strange state of affairs. The first is this: hypermedia (and HTML in particular) hasn't advanced much *since the late 1990s* as hypermedia. Sure, lots of new features have been added to HTML, but there haven't been *any* new ways to interact with a server via pure HTML added in over two decades! HTML developers still only have anchor tags and forms available for building networks interactions, and can only issue `GET` and `POST` requests despite there being many more types of HTTP requests!

This somewhat baffling lack of progress leads immediately to the second and more practical reason that hypermedia has fallen on hard times: as the interactivity and expressiveness of HTML has remained frozen in time, the web itself has marched on, demanding more and more interactive web applications. JavaScript, coupled to data-oriented POJ APIs, has stepped in as a way to provide these new interactive features to end users. It was the *user experience* that you could achieve in JavaScript (and that you couldn't hope to achieve in HTML) that drove the web development community over to the JavaScript-heavy Single Page Application approach.

This is unfortunate, and it didn't have to be this way. There is nothing *intrinsic* to the idea of hypermedia that prevents a richer, more expressive interactivity model. Rather than abandoning the hypermedia architecture, the industry could have demanded more and more interactivity *within* that original, hypermedia model of the web. There is nothing written in stone saying "only forms and anchor elements can interact with a server, and only in response to a few user interactions." JavaScript broke out of this model, why couldn't HTML have done the same? But, reality is what it is: HTML froze in time as a hypermedia and the web development world moved on.

1.2.1. A Hypermedia Comeback?

So, for many developers today working in an industry dominated by JavaScript and SPA frameworks, hypermedia has become an afterthought, or isn't thought about at all. You simply can't get the sort of modern interactivity out of HTML, the hypermedia we all use day to day, necessary for today's modern web applications.

Those of us passionate about hypermedia and the web in general can sit around wishing that, instead of stalling as a hypermedia, HTML had continued to develop, adding new mechanisms for exchanging hypermedia with servers and increasing its general expressiveness. That it was possible to build modern web applications within the original, hypermedia-oriented and REST-ful model that made the early web so powerful, so flexible, so… fun!

In short that hypermedia could, once again, be a legitimate architecture to consider when developing a new web application.

Well, I have some good news. In the last decade, a few idiosyncratic, alternative front end libraries have arisen that attempt to do exactly this! Somewhat ironically, these libraries are all written in JavaScript. However, these libraries use JavaScript not as a *replacement* for the hypermedia architecture, but rather use it to augment HTML itself *as a hypermedia*.

These *hypermedia-oriented* libraries re-center the hypermedia approach as a viable choice for your next web application.

1.2.2. Hypermedia-Oriented Javascript Libraries

In the web development world today there is a debate going on between the SPAs approach and what are now being called "Multi-Page Applications" or MPAs. MPAs are usually just the old, traditional way of building web applications with links and forms across multiple web pages and are thus, by their nature, hypermedia oriented. They are clunky, but, despite this clunkiness, some web developers have become so exasperated at the complexity of SPA applications they have decided to go back to this older way of building things and just accept the limitations of plain HTML.

Some thought leaders in web development, such as Rich Harris, creator of svelte.js, a popular SPA library, propose a mix of the MPA style and the SPA style. Harris calls this approach to building web applications "transitional", in that it attempts to mix both the older MPA approach and the newer SPA approach into a coherent whole, and so is somewhat like the "transitional" trend in architecture, which blends traditional and modern architectural styles. It’s a good term and a reasonable compromise between the two approaches to building web applications.

But it still feels a bit unsatisfactory. Why have two very different architectual models *by default*? Recall that the crux of the tradeoffs between SPAs and MPAs is the *user*

experience or interactivity of the application. This is typically the driving decision when choosing one approach versus the other for an application or, in the case of Transitional Web Applications, for a particular feature.

It turns out that, by adopting a hypermedia oriented library, the interactivity gap closes dramatically between the MPA and SPA approach. You can stay in the simpler hypermedia model for much more of your application, perhaps even all of it. Rather than having an SPA with a bit of hypermedia around the edges, or an even mix of the two dramatically different styles of web development, you can have a web application that is *primarily* hypermedia driven, only kicking out to the more complex SPA approach in the areas that demand it. This can tremendously simplify your web application and provide a much more coherent and understandable final product.

One such hypermedia oriented library is htmx, created by the authors of this book. htmx will be the focus of much (but not all!) of the remainder of this book, and we hope to show you that you can, in fact, create many common "modern" UI features in a web application entirely within the hypermedia model. Not only that, but it is refreshingly fun and simple to do so!

When building a web application with htmx and other hypermedia oriented libraries the term Multi-Page Application applies *roughly*, but it doesn't really capture the crux of the application architecture. htmx, as you will see, does not need to replace entire pages and, in fact, an htmx-based application can reside entirely within a single page. (We don't recommend this practice, but it is certainly possible!)

We rather like to emphasize the *hypermedia* aspect of both the older MPA approach and the newer htmx-based approach. Therefore, we use the term *Hypermedia Driven Applications (HDAs)* to describe both. This clarifies that the core distinction between these approaches and the SPA approach *isn't* the number of pages in the application, but rather the underlying *network* architecture.

So, what would the htmx and, let us say, the HDA equivalent of the JavaScript-based SPA-style button we discussed above look like?

It might look something like this:

Listing 1. 5. an htmx implementation

```
<button hx-get="/contacts" hx-target="#contact-table"> ❶
    Fetch Contacts
</button>
```

❶ An htmx-powered button, issuing a request to `/contacts` and replacing the element with the id `contact-table`

As with the JavaScript example, we can see that this button has been annotated with some attributes. However, in this case we do not have any imperative scripting going on. Instead, we have *declarative* attributes, much like the `href` attribute on anchor tags and the `action` attribute on form tags. The `hx-get` attribute tells htmx: "When the user clicks this button, issue a `GET` request to `/contacts`". The `hx-target` attribute tells htmx: "When the response returns, take the resulting HTML and place it into the element with the id `contact-table`".

I want to emphasize here that the HTTP response from the server is expected to be in *HTML format*, not in JSON. This means that htmx is exchanging *hypermedia* with the server, just like an anchor tag or form might, and thus the interaction is still firmly within this original hypermedia model of the web. htmx *is* adding browser functionality via JavaScript, but that functionality is *augmenting* HTML as a hypermedia, rather than *replacing* the network model with a data-oriented JSON API.

Despite perhaps looking superficially similar to one another, it turns out that this htmx example and the JavaScript-based example are extremely different architectures and approaches to web development. And this generalizes: the HDA approach is also extremely different from the SPA approach.

This may seem somewhat cute: a contrived JavaScript example that no one would ever write in production, and a demo of a small library that perhaps makes HTML a bit more expressive, sure. But this doesn’t look very convincing yet. Sure, this latter approach can’t scale up to large, complex modern web applications and the interactivity that they demand!

In fact, for many applications, it can: just as the original web scaled up surprisingly well via hypermedia, due to the simplicity and flexibility of this approach it *can* often scale extremely well with your application needs. And, despite its simplicity, you will be surprised at just how much we can accomplish in creating modern, sophisticated user

experiences.

1.3. REST

I don't think there is a more misunderstood term in all of software development than REST, which stands for REpresentational State Transfer. You have probably heard this term and, if I asked you which of the two examples, the simple JavaScript button and the htmx-powered button, was REST-ful, there is a good chance you would say that the JavaScript button. It is hitting a JSON data API, and you probably only hear the term REST in the context of JSON APIs! It turns out that this is *exactly backwards*!

It is the *htmx-powered button* that is REST-ful, by virtue of the fact that it is exchanging hypertext with the server.

The industry has been using the term REST largely incorrectly for over a decade now. Roy Fielding, who coined the term REST (and who should know!) had this to say:

> I am getting frustrated by the number of people calling any HTTP-based interface a REST API. Today's example is the SocialSite REST API. That is RPC. It screams RPC. There is so much coupling on display that it should be given an X rating.
>
> What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some broken manual somewhere that needs to be fixed?
>
> — Roy Fielding, https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

We will go into the details of how this happened in a future chapter where we do a deep dive in the famous Chapter 5 of Fielding's PhD dissertation, but for now let me summarize what I view as the crucial practical difference between the two buttons:

In the case of the JavaScript powered button, the client (that is, the JavaScript code) *must understand what a contact is*. It needs to know the internals of the data representation, what is stored where, how to update the data, etc.

In contrast, the htmx-powered button has no knowledge of what a contact it. It simply issues an HTTP request and swaps the resulting HTML into the document. The HTML can change dramatically, introducing or removing all sorts of content and the htmx-button will happly continue exchanging hypermedia with the server. Try changing the content returned

by the JSON API example and see what happens!

This is part of what is called the *Uniform Interface* of REST, and it is the crucial aspect of the hypermedia network architecture that makes it so flexible. Again, we'll talk more about this later, but I wanted to give you a quick peak into *why* hypermedia is so flexible and, I hope, pique your interest in the technical details of the approach for later on in the book.

1.4. When should You Use Hypermedia?

Even if you decide not to use something like htmx and just accept the limitations of plain HTML, there are times when it, and the hypermedia architecture, is worth considering for your project:

Perhaps you are building a web application that simply doesn't *need* a huge amount of user-experience innovation. These are very common and there is no shame in that! Perhaps your application adds its value on the server side, by coordinating users or by applying sophisticated data analysis. Perhaps your application adds value by simply sitting in front of a well-designed database, with simple Create-Read-Update-Delete (CRUD) operations. Again, there is no shame in this!

In any of these cases, using a hypermedia approach would likely be a great choice: the interactivity needs of these applications are not dramatic, and much of the value of the applications live on the server side, rather on than on the client side. They are all amenable to what Roy Fielding, one of the original engineers who worked on the web, called "large-grain hypermedia data transfers": you can simply use anchor tags and forms, with responses that return entire HTML documents from requests, and things will work fine. This is exactly what the web was designed to do.

By adopting the hypermedia approach for these applications, you will save yourself a huge amount of client-side complexity that comes with adopting the Single Page Application approach: there is no need for client-side routing, for managing a client side model, for hand-wiring in JavaScript logic, and so forth. The back button will "just work". Deep linking will "just work". You will be able to focus your efforts on your server, where your application is actually adding value.

Now, by layering htmx or another hypermedia-oriented library on top of this approach, you can address many of the usability issues that come with it by taking advantage of finer-grained hypermedia transfers. This opens up a whole slew of new user interface and

experience possibilities. But more on that later.

1.5. When shouldn’t You Use Hypermedia?

That all being said, and as admitted hypermedia partisans, there are, of course, cases where hypermedia is not the right choice. What would a good example be of such an application?

One example that springs immediately to mind is an online spreadsheet application, where updating one cell could have a large number of cascading changes that need to be made on every keystroke. In this case, we have a highly inter-dependent user interface without clear boundaries as to what might need to be updated given a particular change. Introducing a server round-trip on every cell change would bog performance down terribly! This is simply not a situation amenable to that "large-grain hypermedia data transfer" approach. For an application like this we would certainly look into a sophisticated client-side JavaScript approach.

However, perhaps this online spreadsheet application also has a settings page. And perhaps that settings page *is* amenable to the hypermedia approach. If it is simply a set of relatively straight-forward forms that need to be persisted to the server, the chances are good that hypermedia would, in fact, work great for this part of the app.

And, by adopting hypermedia for that part of your application, you might be able to simplify that part of the application quite a bit. You could then save more of your application’s *complexity budget* for the core, complicated spreadsheet logic, keeping the simple stuff simple. Why waste all the complexity associated with a heavy JavaScript framework on something as simple as a settings page?

What Is A Complexity Budget?

Any software project has a complexity budget, explicit or not: there is only so much complexity a given development team can tolerate and every new feature and implementation choice adds at least a bit more to the overall complexity of the system.

What is particularly nasty about complexity is that it appears to grow exponentially: one day you can keep the entire system in your head and understand the ramifications of a particular change, and a week later the whole system seems intractable. Even worse, efforts to help control complexity, such as introducing abstractions or infrastructure to manage the complexity, often end up making things even more complex. Truly, the job of the good software engineer is to keep complexity under control.

The surefire way to keep complexity down is also the hardest: say no. Pushing back on feature requests is an art and, if you can learn to do it well, making people feel like *they* said no, you will go far.

Sadly this is not always possible: some features will need to be built. At this point the question becomes: "what is the simplest thing that could possibly work?" Understanding the possibilities available in the hypermedia approach will give you another tool in your "simplest thing" tool chest.

This brings up two important points:

First, nearly every SPA application is, at some level, a "Transitional" web application: there is always a bootstrap page that gets the app started that is served via, wait for it, hypermedia! So you are already using the hypermedia approach when you build web applications, whether you think so or not. You are already using HTML in your SPA. Why not make it more expressive and useful?

Second, the hypermedia approach, in both its simple, "vanilla" HTML form and in its more sophisticated forms, can be adopted incrementally: you don’t need to use this approach for your entire application. You can, instead, adopt it where it makes sense. Or, alternatively, you might flip this around and make hypermedia your default approach and only reach for the more complicated JavaScript-based solutions when necessary. We love this latter approach as way to minimize your web applications complexity.

1.6. Summary

- Hypermedia is a unique architecture for building web applications
- Using Data APIs, which is very common in today's web development world, is very dramatically different than the hypermedia approach
- Hypermedia lost out to SPAs & Data APIs due to interactivity limitations, not due to fundamental limitations of the concept
- There is an emerging class of Hypermedia Oriented front-end libraries that recenter hypermedia as the core technology for web development and address these interactivity limitations
- These libraries make Hypermedia Driven Applications (HDAs) a more compelling choice for a much larger set of online applications = Hypermedia In Action :chapter: 2 :sectnums: :figure-caption: Figure 1. :listing-caption: Listing 1. :table-caption: Table 1. :sectnumoffset: 1 :leveloffset: 1 :sourcedir: ../code/src :source-language:

2. 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

2.1. 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.

2.1.1. 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.

2.2. 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("/") ❶
def index(): ❷
    return redirect("/contacts") ❸
```

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

❷ The next method is the handler for that route

❸ 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!

2.3. 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.

2.3.1. 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:

Listing 1. 6. Server Side Search

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

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

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

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

❹ 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' %} ❶

{% block content %} ❷

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

❶ Set the layout template for this template

❷ Delimit the content to be inserted into the layout

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

❹ 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:

Listing 1. 7. The Contacts Table

```
<table>
    <thead>
    <tr>
        <th>First</th> <th>Last</th> <th>Phone</th> <th>Email</th> <th></th>❶
    </tr>
    </thead>
    <tbody>
    {% for contact in contacts %} ❷
        <tr>
            <td>{{ contact.first }}</td>
            <td>{{ contact.last }}</td>
            <td>{{ contact.phone }}</td>
            <td>{{ contact.email }}</td> ❸
            <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
                <a href="/contacts/{{ contact.id }}">View</a></td> ❹
        </tr>
    {% endfor %}
    </tbody>
</table>
```

❶ We output some headers for our table

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

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

❹ 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 link 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> ❶
    </p>

{% endblock %} ❷
```

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

❷ 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!

2.3.2. 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']) ❶
def contacts_new_get():
    return render_template("new.html", contact=Contact()) ❷
```

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

❷ 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"> ❶
    <fieldset>
        <legend>Contact Values</legend>
        <p>
            <label for="email">Email</label> ❷
            <input name="email" id="email" type="email" placeholder="Email" value="{{
contact.email or '' }}"> ❸
            <span class="error">{{ contact.errors['email'] }}</span> ❹
        </p>
```

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

❷ A label for the first form input

❸ the first form input, of type email

❹ 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']) ❶
    if c.save(): ❷
        flash("Created New Contact!")
        return redirect("/contacts") ❸
    else:
        return render_template("new.html", contact=c) ❹
```

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

❷ We try to save it

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

❹ 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!

2.3.3. 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>") ❶
def contacts_view(contact_id=0): ❷
    contact = Contact.find(contact_id) ❸
    return render_template("show.html", contact=contact) ❹
```

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

❷ The handler takes the value of this path parameters

❸ Look up the corresponding contact

❹ 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.

2.3.4. 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!

2.3.5. 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"> ❶
        <fieldset>
            <legend>Contact Values</legend>
              <p>
                  <label for="email">Email</label>
                  <input name="email" id="email" type="text" placeholder="Email"
value="{{ contact.email }}"> ❷
                  <span class="error">{{ contact.errors['email'] }}</span>
              </p>
```

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

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

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

❷ Look the contact up by id

❸ update the contact with the new information from the form

❹ Attempt to save it

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

❻ 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!

2.3.6. 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"]) ❶
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete() ❷
    flash("Deleted Contact!")
    return redirect("/contacts") ❸
```

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

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

❸ 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!

2.3.7. 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.

2.4. 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 = Hypermedia In Action :chapter: 3 :sectnums: :figure-caption: Figure

1. :listing-caption: Listing 1. :table-caption: Table 1. :sectnumoffset: 2 :leveloffset: 1 :sourcedir: ../code/src :source-language:

3. Extending HTML As Hypermedia

This chapter covers:

- The shortcomings of "plain" HTML
- How htmx addresses these shortcomings
- How to issue various HTTP requests with htmx
- History and back button support in htmx

3.1. The Shortcomings of "Plain" HTML

In the previous chapter we introduced a simple Web 1.0-style hypermedia application to manage contacts. This application supported the normal CRUD operations for contacts, as well as a simple mechanism for searching contacts. Our application was built using nothing but forms and anchor tags, the traditional tags used to interact with servers, and it exchanges hypermedia (HTML) with the server over HTTP, issuing `GET` and `POST` HTTP requests and receiving back full HTML documents in response. It is pretty simple, but it is also definitely a Hypermedia Driven Application.

Our application is robust, leverages the web's strengths and is simple to understand. So what's not to like? Well, unfortunately, our application isn't completely satisfying from either a user experience perspective, or from a technical perspective. It suffers from problems typical of this style of Web 1.0 applications.

Two obvious problems that jump out are:

- From a user experience perspective: there is a noticeable refresh when you move between pages of the application, of when you create, update or delete a contact. This is because every user interaction (link click or form submission) requires a full page refresh, with a whole new HTML document to process after each action.
- From a technical perspective, all the updates are done with the `POST` HTTP action. This is despite the fact that more logical actions HTTP request types like `PUT` and `DELETE` exist and would make far more sense for some of the operations. Somewhat ironically, since we are using pure HTML, we are unable to access the full expressive power of

HTTP!

The first point, in particular, is noticeable in Web 1.0 style applications like ours and is what is responsible for giving them the reputation for being "clunky" when compared with their more sophisticated JavaScript-based Single Page Application cousins.

Single Page Applications eliminate this clunkiness by updating a web page directly, mutating the Document Object Model (DOM), the JavaScript API to the underlying HTML page. There are a few of different styles of SPA, but, as we discussed in Chapter 1, the most common today is to tie the DOM to a JavaScript model and let an SPA framework like react *reactively* update the DOM when the JavaScript model is updated: you make a change to a JavaScript object and the web page magically updates its state to reflect the change in the model.

Recall that in this style of application communication with the server is typically done via a JSON Data API, with the application sacrificing the advantages of hypermedia in order to provide a better, smoother user experience.

Many web developers today would not even consider the hypermedia approach due to the perceived "legacy" feel of these Web 1.0 style applications.

The second, technical point may strike you as a bit pedantic, and I am the first to admit that conversations around REST and which HTTP Action is right for a given operation can become very tedious. But, nonetheless, it has to be admitted that, when using plain HTML, it is impossible to use HTTP to its full power and, therefore, it is impossible to realize the full vision of the web as a REST-ful system: a complete, stateless, resource-oriented distributed networking architecture that is flexible and resilient.

3.1.1. A Close Look At A Hyperlink

As we have been saying, it turns out that you can actually get a lot of interactivity out of the hypermedia model, if you adopt a hypermedia-oriented library like htmx. To understand conceptually how htmx allows us to better address the UX concerns of Web 1.0 style applications, let’s revisit the hyperlink/anchor tag from Chapter 1 and really drill in to each facet of it:

This simple anchor tag, when interpreted by a browser, creates a hyperlink to the Manning website:

Listing 1. 8. A Simple Hyperlink, Again

```
<a href="https://www.manning.com/">
  Manning Books
</a>
```

Breaking down exactly what this link will tell the browser to do, we have the following list:

- The browser will render the text "Manning Books" to the screen, likely with a decoration indicating it is clickable
- Then, when a user clicks on the text…
- The browser will issue an HTTP `GET` to `https://www.manning.com` and then…
- The browser will load the HTTP response into the browser window, replacing the current document

So we have four aspects of a simple hypermedia link like this, with the last three being the mechanic that distinguishes a hyperlink from "normal" text.

Let's take a moment and think about how we can generalize this fundamental hypermedia mechanic of HTML. There is no rule saying that hypermedia can *only* work this way, after all!

An initial observation is: why are anchor tags so special? Shouldn't other elements (besides forms) be able to issue HTTP requests as well? For example, shouldn't `button` elements be able to do so? It seemed silly to have to wrap a form tag around a button to make deleting contacts work in our application. Why should only anchor tags and forms be able to issue requests?

This presents our first opportunity to expand the expressiveness of HTML: we can allow *any* element to issue a request to the server.

For our next observation, let's consider the event that triggers the request to the server on our link: a click. Well, what's so special about clicking (in the case of anchors) or submitting (in the case of forms)? Those are just two of many, many events that are fired by the DOM, after all. Events like mouse down, or key up, or blur are all events you might want to use to issue an HTTP request. Why shouldn't these other events be able to trigger requests as well?

This gives us our second opportunity to expand the expressiveness of HTML: we can allow *any* event, not just a click, as in the case of our hyperlink, to trigger an HTTP request.

Getting a bit more technical in our thinking leads us to the problem we noted earlier in the chapter: plain HTML only give us access to the `GET` and `POST` actions of HTTP? HTTP *stands* for HyperText Transfer Protocol, and yet the format it was explicitly designed for, HTML, only supports two of the five developer-facing request types! You *have* to use JavaScript and issue an AJAX request to get at the other three: `DELETE`, `PUT` and `PATCH`.

Let's recall what are all of these different HTTP request types designed to represent?

- `GET` corresponds with "getting" a representation for a resource from a URL: it is a pure read, with no mutation of the resource
- `POST` submits an entity (or data) to the given resource, often creating or mutating the resource and causing a state change
- `PUT` submits an entity (or data) to the given resource for update or replacement, again likely causing a state change
- `PATCH` is similar to `PUT` but implies a partial update rather than a complete replacement of the entity
- `DELETE` deletes the given resource

These operations correspond closely to the CRUD operations we discussed in Chapter 2, and by only giving us access to two of them, HTML is presenting us with a severe and obvious technical limitation.

So here is our third opportunity to expand the expressiveness of HTML: we can allow HTML to have access to the missing three HTTP actions, `PUT`, `PATCH` and `DELETE`.

As a final observation, consider that last aspect of a hyperlink: it replaces the *entire* screen when a user clicks on it. It is this technical detail that makes for a poor user experience: it causes flashes of unstyled content, a loss of scroll state and so forth. But, again, there is no rule saying that hypermedia exchanges *must* replace the entire document.

This gives us our forth, final and perhaps most important opportunity to generalize HTML: what if we allowed the hypermedia response to replace elements *within* the current document, rather than requiring that it replace the entire document. This would make

Hypermedia Driven Applications function much more like a Single Page Application, where only part of the DOM is updated by a given user interaction or network request.

If we were to take these four opportunities to generalize HTML, we would be extending HTML far beyond its normal capabilities, and we would be doing so *entirely within* the normal, hypermedia model of the web. Note that none of the extensions involve going outside the normal exchanging-HTML-over-HTTP found in Web 1.0 applications. Rather, the all four are simply generalizations of existing functionality already found within HTML.

3.2. Extending HTML as a Hypermedia with htmx

It turns out that there are some JavaScript libraries that extends HTML in exactly this manner. This may seem somewhat ironic, given that JavaScript-based SPAs have supplanted HTML-based hypermedia applications, that JavaScript would be used in this manner. But JavaScript is simply a language for extending browser functionality on the client side, and there is no rule saying it has to be used to write SPAs. In fact, JavaScript is the perfect tool for addressing the shortcomings of HTML as a hypermedia!

One such library is htmx, which will be the focus of the next few chapters. htmx is not the only JavaScript library that takes this hypermedia-oriented approach, but it is perhaps the purest in the pursuit of extending HTML as a hypermedia. It focuses intensely on the four limitations discussed above and attempts to incrementally address each one, without introducing a significant amount of additional conceptual infrastructure for web developers.

3.2.1. Installing and Using htmx

From a practical, getting started perspective, htmx is a simple, dependency-free and stand-alone library that can be added to a web application by simply including it via a `script` tag in your `head` element

Because of this simple installation model, we can take advantage of tools like public CDNs to install the library. Below we are using the popular unpkg Content Delivery Network (CDN) to install version `1.7.0` of the library. We use an integrity hash to ensure that the delivered content matches what we expect. This SHA can be found on the htmx website. Finally, we mark the script as `crossorigin="anonymous"` so no credentials will be sent to the CDN.

Listing 1. 9. Installing htmx

```
<head>
  <script src="https://unpkg.com/htmx.org@1.7.0"
          integrity="sha384-
EzBXYPt0/T6gxNp0nuPtLkmRpmDBbjg6WmCUZRLXBBwYYmwAUxzlSGej0ARHX0Bo"
          crossorigin="anonymous"></script>

</head>
```

Believe it or not, that's all it takes to install htmx! If you are used to the extensive build systems in today's JavaScript world, this may seem impossible or insane, but this is in the spirit of the early web: you could simply include a script tag and things would just work. And it still feels like magic, even today!

Of course, you may not want to use a CDN, in which case you can download htmx to your local system and adjust the script tag to point to wherever you keep your static assets. Or, you may have one of those more sophisticated build system that automatically installs dependencies. In this case you can use the Node Package Manager (npm) name for the library: `htmx.org` and install it in the usual manner that your build system supports.

Once htmx has been installed, you can begin using it immediately.

And here we get to the funny part of htmx: unlike the vast majority of JavaScript libraries, htmx does not require you, the user, to actually write any JavaScript!

Instead, you will use *attributes* placed directly on elements in your HTML to drive more dynamic behavior. Remember: htmx is extending HTML as a hypermedia, and we want that extension to be as natural and consistent as possible with existing HTML concepts. Just as an anchor tag uses an `href` attribute to specify the URL to retrieve, and forms use an `action` attribute to specify the URL to submit the form to, htmx uses HTML *attributes* to specify the URL that an HTTP request should be issued to.

3.3. Triggering HTTP Requests

Let's look at the first feature of htmx: the ability for any element in a web page to issue HTTP requests. This is the core functionality of htmx, and it consists of five attributes that can be used to issue the five different developer-facing types of HTTP requests:

- `hx-get` - issues an HTTP `GET` request
- `hx-post` - issues an HTTP `POST` request
- `hx-put` - issues an HTTP `PUT` request
- `hx-patch` - issues an HTTP `PATCH` request
- `hx-delete` - issues an HTTP `DELETE` request

Each of these attributes, when placed on an element, tell the htmx library: "When a user clicks (or whatever) this element, issue an HTTP request of the specified type"

The values of these attributes are similar to the values of both `href` on anchors and `action` on forms: you specify the URL you wish to issue the given HTTP request type to. Typically, this is done via a server-relative path.

So, for example, if we wanted a button to issue a `GET` request to `/contacts` then we would write:

Listing 1. 10. A Simple htmx-Powered Button

```
<button hx-get="/contacts"> ❶
  Get The Contacts
</button>
```

❶ A simple button that issues an HTTP `GET` to `/contacts`

htmx will see the `hx-get` attribute on this button, and hook up some JavaScript logic to issue an HTTP `GET` AJAX request to the `/contacts` path when the user clicks on it. Very easy to understand and very consistent with the rest of HTML.

3.3.1. It's All Just HTML!

Now we get to perhaps the most important thing to understand about htmx: it expects the response to this AJAX request *to be HTML*! htmx is an extension of HTML and, just as the response to an anchor tag click or form submission is usually expected to be HTML, htmx expects the server to respond with a hypermedia, namely with HTML.

This may come as a shock to web developers who are unused to responding to an AJAX request with anything other than JSON, which is far and away the most common response format for such requests. But AJAX requests are just HTTP requests and there is no rule

saying they must be JSON! Recall again that AJAX stands for Asynchronous Javascript & XML, so JSON is already a step away from the format originally envisioned for this API: XML. htmx simply goes another direction and expects HTML.

htmx vs. "plain" HTML responses

So, we have established that htmx expects HTML responses to the HTTP requests it makes. But there is an important difference between the HTTP responses to normal anchor and form driven requests and to htmx-powered requests like the one made by this button: in the case of htmx triggered requests, responses are often only *partial* bits of HTML.

In htmx-powered interactions we are typically not replacing the entire document, so it is not necessary to transfer an entire HTML document from the server to the browser. This fact can be used to save bandwidth as well as resource loading time, since less overall content is transferred from the server to the client and since it isn't necessary to reprocess a `head` tag with style sheets,script tags, and so forth.

A simple *partial* HTML response to the button's htmx request might look like this:

Listing 1. 11. A partial HTML Response to an htmx Request

```
<ul>
  <li><a href="mailto:joe@example.com">Joe</a></li>
  <li><a href="mailto:sarah@example.com">Sarah</a></li>
  <li><a href="mailto:fred@example.com">Fred</a></li>
</ul>
```

This is just a simple unordered list of contacts with some clickable elements in it. Note that there is no opening `html` tag, no `head` tag, and so forth: it is a raw HTML list, without any decoration around it. A response in a real application might of course contain far more sophisticated HTML than a simple list, but it wouldn't need to be an entire page of HTML.

This response is perfect for htmx: it will take the returned content and swap it in to the DOM. This is fast and efficient, leveraging the existing HTML parser in the browser. And this demonstrates that htmx is staying within the hypermedia paradigm: just like in a "normal" web application, we see hypermedia being transferred to the client in a stateless and uniform manner, where the client knows nothing about the internals of the resources

being displayed.

This button just a more sophisticated component for building a Hypermedia Driven Application!

3.4. Targeting Other Elements

Now, given that htmx has issued a request and gotten back some HTML as a response, what should be done with it?

It turns out that the default htmx behavior is to simply put the returned content inside the element that triggered the request. That's obviously *not* a good thing in this situation: we will end up with a list of contacts awkwardly embedded within a button element on the page! That will look pretty silly and is obviously not what we want.

Fortunately htmx provides another attribute, `hx-target` which can be used to specify exactly where in the DOM the new content should be placed. The value of the `hx-target` attribute is a Cascading Style Sheet (CSS) *selector* that allows you to specify the element to put the new hypermedia content into

Let's add a `div` tag that encloses the button with the id `main`. We will then target this `div` with the response:

Listing 1. 12. A Simple htmx-Powered Button

```
<div id="main"> ❶

  <button hx-get="/contacts" hx-target="#main"> ❷
    Get The Contacts
  </button>

</div>
```

❶ A `div` element that wraps the button

❷ A new `hx-target` attribute that specifies the `div` as the target of the response

We have added `hx-target="#main"` to our button, where `#main` is a CSS selector that says "The thing with the ID 'main'". Note that by using CSS selectors, htmx is once again building on top of familiar and standard HTML concepts. By doing so it keeps the additional conceptual load beyond normal HTML to a minimum.

Given this new configuration, what would the HTML on the client look like after a user clicks on this button and a response has been received and processed?

It would look something like this:

Listing 1. 13. Our HTML After the htmx Request Finishes

```
<div id="main">
  <ul>
    <li><a href="mailto:joe@example.com">Joe</a></li>
    <li><a href="mailto:sarah@example.com">Sarah</a></li>
    <li><a href="mailto:fred@example.com">Fred</a></li>
  </ul>
</div>
```

The response HTML has been swapped into the `div`, replacing the button that triggered the request. This all has happened "in the background" via AJAX, without a large page refresh. Nonetheless, this is *definitely* a hypermedia interaction. It isn't as coarse-grained as a normal, full web page request coming from an anchor might be, but it certainly falls within the same conceptual model!

3.5. Swap Styles

Now, maybe we don't want to simply load the content from the *into* the div. Perhaps, for whatever reasons, we wish to *replace* the entire div with the response.

htmx provides another attribute, `hx-swap`, that allows you to specify exactly *how* the content should be swapped into the DOM. (Are you beginning to sense a pattern here?) The `hx-swap` attribute supports the following values:

- `innerHTML` - The default, replace the inner html of the target element
- `outerHTML` - Replace the entire target element with the response
- `beforebegin` - Insert the response before the target element
- `afterbegin` - Insert the response before the first child of the target element
- `beforeend` - Insert the response after the last child of the target element
- `afterend` - Insert the response after the target element

- `delete` - Deletes the target element regardless of the response
- `none` - No swap will be performed

The first two values, `innerHTML` and `outerHTML`, are taken from the standard DOM properties that allow you to replace content within an element or in place of an entire element respectively.

The next four values are taken from the `Element.insertAdjacentHTML()` DOM API, which allow you to place an element or elements around a given element in various ways.

The last two values, `delete` and `none` are specific to htmx, but should be fairly obvious for you understand.

Again, you can see that htmx tries to stay as close as possible to the existing web standards to keep your conceptual load to a minimum.

Let's consider if, rather than replacing the `innerHTML` content of the main div above, we wished to replace the *entire div* with the HTML response. To do so would require only a small change to our button:

Listing 1. 14. Replacing the Entire div

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML"> ❶
    Get The Contacts
  </button>

</div>
```

❶ The `hx-swap` attribute specifies how to swap new content in

Now, when a response is received, the *entire* div will be replaced with the hypermedia content:

Listing 1. 15. Our HTML After the htmx Request Finishes

```
<ul>
  <li><a href="mailto:joe@example.com">Joe</a></li>
  <li><a href="mailto:sarah@example.com">Sarah</a></li>
  <li><a href="mailto:fred@example.com">Fred</a></li>
</ul>
```

You can see that, with this change, the target div has been entirely removed from the DOM, and the list that was returned as the response has replaced it.

Later in the book we will see additional uses for `hx-swap`, for example when we implement infinite scrolling in our contact management application.

Note that with the `hx-get`, `hx-post`, `hx-put`, `hx-patch` and `hx-delete` attributes, we have addressed two of the shortcomings that we enumerated regarding plain HTML: we can now issue an HTTP request with *any* element (in this case we are using a button). Additionally, we can issue *any sort* of HTTP request we want, `PUT`, `PATCH` and `DELETE`, in particular.

And, with `hx-target` and `hx-swap` we have addressed a third shortcoming: the requirement that the entire page be replaced. Now we have the ability, within our hypermedia, to replace any element we want and in any manner we wish to replace it.

So, with seven relatively simple additional attributes, we have addressed most of the hypermedia shortcomings we identified earlier with HTML. Not bad!

There was one remaining shortcoming of HTML that we noted: the fact that only a `click` event (on an anchor) or a `submit` event (on a form) can trigger HTTP request. Let's look at how we can address that concern next.

3.6. Using Other Events

Thus far we have been using a button to issue a request with htmx. You have probably intuitively understood that the request will be issued when the button is clicked on since, well, that's what you do with buttons! You click on them!

And, yes, by default when an `hx-get` or another request-driving annotation from htmx is placed on a button, the request will beissued when the button is clicked.

However, htmx generalizes this notion of an event triggering a request by using, you guessed it, another attribute: `hx-trigger`. The `hx-trigger` attribute allows you to specify one or more events that will cause the element to trigger an HTTP request, overriding the default triggering event.

What is the "default triggering event" in htmx? It depends on the element type, but should be fairly intuitive to anyone familiar with HTML:

- Requests on `input`, `textarea` & `select` elements are triggered by the `change` event
- Requests on `form` elements are triggered on the `submit` event
- Requests on all other elements are triggered by the `click` event

So, lets consider if we wanted to trigger the request on our button when the mouse entered it. This is certainly not a recommended UX pattern, but let's just look at it as an example!

To do this, we would add the following attribute to our button:

Listing 1. 16. A Terrible Idea, But It Demonstrates The Concept!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="mouseenter"> ❶
    Get The Contacts
  </button>

</div>
```

❶ Issue a request… on the `mouseenter` event?

Now, whenever the mouse enters this button, a request will be triggered. Hey, we didn't say this was a *good* idea!

Let's try something a bit more realistic: let's add support for a keyboard shortcut for loading the contacts, `Ctrl-L` (for "Load"). To do this we will need to take advantage of some additional syntax that the `hx-trigger` attribute supports: event filters and additional arguments.

Event filters are a mechanism for determining if a given event should trigger a request or

not. They are applied to an event by adding square brackets after it: `someEvent[someFilter]`. The filter itself is a JavaScript expression that will be evaluated when the given event occurs. If the result is truthy, in the JavaScript sense, it will trigger the request. If not, it will not.

In the case of keyboard shortcuts, we want to catch the `keyup` event in addition to the keyup event:

Listing 1. 17. A Start

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup"> ❶
    Get The Contacts
  </button>

</div>
```

❶ A trigger with two events

Note that we have a comma separated list of events that can trigger this element, allowing us to respond to more than one potential triggering event.

There are two problems with this:

- It will trigger requests on *any* keyup event
- It will trigger requests only when a keyup occurs *within* this button (an unlikely occurrence!)

To fix the first issue, lets use a trigger filter:

Listing 1. 18. Better!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup[ctrlKey && key == 'l']"> ❶
    Get The Contacts
  </button>

</div>
```

❶ A trigger with an added filter, specifying that the control key and L must be pressed

The trigger filter in this case is `ctrlKey && key == 'l'`. This can be read as "A key up event, where the ctrlKey property is true and the key property is equal to 'l'". Note that the properties `ctrlKey` and `key` are resolved against the event rather than the global name space, so you can easily filter on the properties of a given event. You can use any expression you like for a filter, however: calling a global JavaScript function, for example, is perfectly acceptable.

OK, so this filter limits the keyups that will trigger the request to only `Ctrl-L` presses. However, we still have the problem that, as it stands, only `keyup` events *within* the button will trigger the request. If you are familiar with the JavaScript event bubbling model: events typically "bubble" up to parent elements so an event like a keyup will be triggered first on the focused element, then on it's parent, and so on, until it reaches the top level `document` that is the root of all other elements.

In this case, this is obviously not what we want! People typically aren't typing characters *within* the button, they click on buttons! Here we want to listen to the `keyup` events on the entire page, or, equivalently, on the `body` element.

To fix this, we need to take advantage of another feature that the `hx-trigger` attribute supports: the ability to listen to *other elements* for events using the `from:` modifier. The `from:`modifier, as with many other attributes and modifiers in htmx, uses a CSS selector to select the element to listen on.

We can use it like this:

Listing 1. 19. Better!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup[ctrlKey && key == 'L'] from:body">❶
    Get The Contacts
  </button>

</div>
```

❶ Listen to the event on the `body` tag

Now, in addition to clicks, our button is listening for `keyup` events on the body of the page, and should issue a request both when it is clicked on, and also whenever someone hits `Ctrl-L` within the body of the page!

A nice little keyboard shortcut! Perfect!

The `hx-trigger` attribute is more elaborate than the other htmx attributes we have looked at so far, but that is because events, in general, are used more elaborately in modern user interfaces. The default options often suffice, however, and you shouldn't need to reach for complicated trigger features too often when using htmx.

That being said, even in the more elaborate situations like the example above, where we have a keyboard shortcut, the overall feel of htmx is *declarative* rather than *imperative* and follows along closely with the standard feel and philosophy of HTML.

And hey, check it out! With this final attribute, `hx-trigger`, we have addressed *all* of the shortcomings of HTMl that we enumerated at the start of this chapter. That's a grand total of eight, count 'em, *eight* attributes that all fall squarely within the same conceptual model as normal HTML and that, by extending HTML as a hypermedia, open up world of new user interface possibilities!

3.7. Passing Request Parameters

So far we have been just looking at situation where a button makes a simple `GET` request. This is conceptually very close to what an anchor tag might do. But there is the other primary element in traditional hypermedia-based applications: forms. Forms are used to pass additional information beyond just a URL up to the server in a request. This

information is typically entered into elements within the form via the various types of input tags in HTML.

htmx allows you include this additional information in a natural way that mirrors how HTML itself works.

3.7.1. Enclosing Forms

The simplest way to pass additional input values up with a request in htmx is to enclose the input within a form tag.

Let's take our original button for retrieving contacts and repurpose it for searching contacts:

Listing 1. 20. A Simple htmx-Powered Button

```
<div id="main">

  <form> ❶
      <label for="search">Search Contacts:</label>
      <input id="search" name="q" type="search" placeholder="Search Contacts"> ❷
      <button hx-post="/contacts" hx-target="#main"> ❸
        Search The Contacts
      </button>
  </form>

</div>
```

❶ The form tag encloses the button, thereby including all values within it in the button request

❷ A new input that users will be able to enter search text into

❸ Our button has been converted to an `hx-post`

Here we have added a form tag surrounding the button along with a search input that can be used to enter a term to search the contacts with.

Now, when a user clicks on the button, the value of the input with the id `search` will be included in the request. This is by virtue of the fact that there is a form tag enclosing both the button and the input: when an htmx-driven request is triggered, htmx will look up the DOM hierarchy for an enclosing form, and, if one is found, it will include all values from within that form. (This is sometimes referred to as "serializing" the form.)

You might have noticed that the button was switched from a `GET` request to a `POST` request. This is because, by default, htmx does *not* include the closest enclosing form for `GET` requests. This is to avoid serializing forms in situations where the data is not needed and to keep URLs clean when dealing with history entries, which we discuss in the next section.

3.7.2. Including inputs

While enclosing all the inputs you want included in a request is the most common approach for including values from inputs in htmx requests, it isn't always ideal: form tags have layout consequences and cannot be placed in some places (forms, for example). So htmx provides another mechanism for including value in requests: the `hx-include` attribute which allows you to select input values that you wish to include in a request via CSS selectors.

Here is the above example reworked to include the input, dropping the form:

Listing 1. 21. A Simple htmx-Powered Button

```
<div id="main">

  <label for="search">Search Contacts:</label>
  <input id="search" name="q" type="search" placeholder="Search Contacts">
  <button hx-post="/contacts" hx-target="#main" hx-include="#search">❶
    Search The Contacts
  </button>

</div>
```

❶ `hx-include` can be used to include values directly in a request

The `hx-include` attribute takes a CSS selector value and allows you to specify exactly which values to send along with the request. This can be useful if it is difficult to colocate an element issuing a request with all the inputs that need to be submitted with it. It is also useful when you do, in fact, want to submit values with a `GET` request and overcome the default behavior of htmx with respect to `GET` requests.

3.7.3. Inline Values

A final way to include values in htmx-driven requests is to use the `hx-vals` attribute, which allows you to include static JSON-based values in the request. This can be useful if you have additional context you wish to encode during server side rendering for a request.

Here is an example:

Listing 1. 22. A Simple htmx-Powered Button

```
<button hx-get="/contacts" hx-vals='{"state":"MT"}'> ❶
  Get The Contacts In Montana
</button>
```

❶ `hx-vals`, a JSON value to include in the request

The parameter `state` the value `MT` will be included in the `GET` request, resulting in a path and parameters that looks like this: `/contacts?state=MT`. One thing to note is that we switched the `hx-vals` attribute to use single quotes around its value. This is because JSON strictly requires double quotes and, therefore, to avoid escaping we needed to use the sinqle-quote form for the attribute value.

This approach is useful when you have fixed data that you want to include in a request and you don't want to rely on something like a hidden input. You can also prefix `hx-vals` with a `js:` and pass values evaluated at the time of the request, which can be useful for including things like a dynamically maintained variable, or value from a third party javascript library.

These three mechanisms allow you to include values in your hypermedia requests with htmx in a manner that is very familiar and in keeping with the spirit of HTML.

3.8. History Support

A final piece of functionality to discuss to close out our overview of htmx is browser history. When you use normal HTML links and forms, your browser will keep track of all the pages that you have visited. You can use the back button to navigate back to a previous page and, once you have done this, you can use a forward button to go forward to the original page you were on.

This notion of history was one of the killer features of the early web. Unfortunately it turns out that history becomes tricky when you move to the Single Page Application paradigm. An AJAX request does not, by itself, register a web page in your browsers history and this is a good thing! An AJAX request may have nothing to do with the state of the web page (perhaps it is just recording some activity in the browser), so it wouldn't be appropriate to create a new history entry for the interaction.

However, there are likely to be a lot of AJAX driven interactions in a Single Page Application where it *is* appropriate to create a history entry. And JavaScript does provide an API for working with the history cache. Unfortunately the API is very difficult to work with and is often simply ignored by developers. If you have ever used a Single Page Application and accidentally clicked the back button, only to lose your entire application state and have to start over, you have seen this problem in action.

In htmx, as in Single Page Application frameworks, you often need to explicitly work with the history API. Fortunately, htmx makes it much easier to do so than most other libraries.

Consider the button we have been discussing again:

Listing 1. 23. Our trusty button

```
<button hx-get="/contacts" hx-target="#main">
  Get The Contacts
</button>
```

As it stands, if you click this button it will retrieve the content from `/contacts` and load it into the element with the id `main`, but it will *not* create a new history entry. If we wanted it to create a history entry we would add another attribute to the button, `hx-push-url`:

Listing 1. 24. Our trusty button, now with history!

```
<button hx-get="/contacts" hx-target="#main" hx-push-url="true">❶
  Get The Contacts
</button>
```

❶ `hx-push-url` will create an entry in history when the button is clicked

Now, when the button is clicked, the `/contacts` path will be put into the browser's navigation bar and a history entry will be created for it. Furthermore, if the user clicks the back button, the original content for the page will be restored, along with th original URL.

`hx-push-url` might sound a little obscure, but this is based on the JavaScript API, `history.pushState()`. This notion of "pushing" derives from the fact that history entries are modeled as a stack, and so you are "pushing" new entries onto the top of the stack of history entries.

With this (relatively) simple mechanism, htmx allows you to integrate with the back button in a way that mimics the "normal" behavior of HTML. Not bad if you look at what other javascript librarires require of you!

Drawbacks To The htmx Approach

htmx is a very pure extension to HTML, aiming to incrementally improve the language as a hypermedia in a manner that is conceptually coherent with the underlying markup language. This approach, like any technical choice, is not without tradeoffs: by staying so close to HTML, htmx does not give developers a lot of infrastructure that many might feel should be there "by default".

A good example is the concept of modal dialogs. Many web applications today make heavy use of modal dialogs, effectively in-page pop-ups that sit "on top" of the existing page. (Of course, in reality, this is an optical illusion and it is all just a web page: the web has no notion of"modals" in this regard.)

A web developer might expect htmx to provide some sort of modal dialog component out of the box, since it is, after all, a front-end library, and many front end libraries offer support for this pattern.

htmx, however, has no notion of modals. That’s not to say you can’t use modals with htmx, and we will look at how you can do so later. But htmx, like HTML itself, won’t give you an API specifically for creating modals. You would need to use a 3rd party library or roll your own modal implementation and then integrate htmx into it if you want to use modals within an htmx-based application.

This is the design tradeoff that htmx makes: it retains conceptual purity as an extension of HTML, and, in exchange, lacks some of the "batteries included" features found in other front end libraries.

As an aside, it’s worth nothing that htmx *can* be used to effectively implement a slightly different UX pattern, inline editing, which is often a good alternative to modals, and, in our opinion, is more consistent with the stateless nature of the web.

3.9. Summary

- Unfortunately, HTML has some shortcomings as a hypermedia:
  - It doesn't give you access to non-`GET` or `POST` requests
  - It requires that you update the entire page
  - It only offers limited interactivity with the user
- htmx addresses each of these shortcomings, increasing the expressiveness of HTML as a hypermedia
- The `hx-get`, `hx-post`, etc. attributes can be used to issue requests with any element in the dom
- The `hx-swap` attribute can be used to control exactly how HTML responses to htmx requests should be swapped into the DOM
- The `hx-trigger` attribute can be used to control the event that triggers a request
- Event filters can be used in `hx-trigger` to narrow down the exact situation that you want to issue a request for
- htmx offers three mechanisms for including additional input information with requests:
  - Enclosing elements within a `form` tag
  - Using the `hx-include` attribute to select inputs to include in the request
  - `hx-vals` for embedding values directly via JSON or, dynamically, resolving values via JavaScript
- htmx also provides integration with the browser history and back button, using the `hx-push-url` attribute = Hypermedia In Action :chapter: 4 :sectnums: :figure-caption: Figure 1. :listing-caption: Listing 1. :table-caption: Table 1. :sectnumoffset: 3 :leveloffset: 1 :sourcedir: ../code/src :source-language:

4. Putting Hypermedia Into Action With htmx

This chapter covers

- Installing htmx in our application
- Adding AJAX-based navigation to our application via "boosting"
- Implementing a proper delete mechanic for contacts

- Validating emails as the user types
- Implementing paging in our application
- Implementing the "Click To Load" pattern
- Implementing the "Infinite Scroll" pattern

4.1. Installing htmx

Now that we've seen how htmx extends HTML as a hypermedia, it's time to put it into action. We will still be exchanging hypermedia, that is HTML, with our server, but we will have a more powerful hypermedia to work with. This will allow us to address user experience issues, such as long feedback cycles or painful page refreshes, without needing to write much, if any, JavaScript, without creating an elaborate JSON API, and so on: everything will be implemented in hypermedia, using the core concepts of the original web.

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

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

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

Listing 1. 25. Installing htmx

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

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

4.2. Adding AJAX Navigation

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

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

4.2.1. Boosted Links

Let's take a look at an example of a boosted link. Here we have a link to a hypothetical settings page. Because it has `hx-boost="true"` on it, htmx will prevent the normal link behavior of issuing a request to the `/settings` path and replacing the entire page with the response. Instead, htmx will issue an AJAX request to `/settings`, taking the result and replacing the `body` element with the new content.

Listing 1. 26. A Boosted Link

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

❶ A simple attribute makes this link AJAX-powered

Now, you might wonder: what's the advantage here? We are issuing an AJAX request and simply replacing the entire body.

Is that significantly different from just issuing a normal link request?

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

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

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

4.2.2. Boosted Forms

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

Here is an example of a form that posts messages to the `/messages` end point using an HTTP `POST` request. By adding `hx-boost` to it, those requests will be done in AJAX, rather than the normal browser behavior.

Listing 1. 27. A Boosted Form

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

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

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

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

4.2.3. Attribute Inheritance

Let's expand on our previous example of a boosted link, and add a few more boosted links along side it. We add links so we ahve a link to the `/contacts` page, the `/settings` page, and the `/help` page. All these links are boosted and will behave in the manner that we have described.

But this feels a little redundant, doesn't it? It is a shame we have to annotate three links with the `hx-boost="true"` right next to one another.

Listing 1. 28. A Set of Boosted Links

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

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

So to avoid the redundancy in this example, lets introduce a `div` element that encloses all the links and "hoist" the `hx-boost` attribute up to it. This will let us remove the redundant `hx-boost` attributes, but ensure all the links are still boosted, inheriting that functionality from the parent `div`. Note that any legal element type could be used here, we just used a `div` out of habit.

1. Boosting Links Via The Parent

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

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

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

link to behave normally.

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

1. Boosting Links Via The Parent

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

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

❷ The boosting behavior is overridden for this link

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

4.2.4. Progressive Enhancement

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

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

This is not to say that htmx *always* offers progressive enhancement. It is certainly possible to build features that do not offer a "No JS" fallback in htmx, and, in fact, many of the features we will build later in the book will fall into this category. (I will note when a

feature is progressive enhancement friendly and when it is not.) Ultimately, it is up to you, the developer, to decide if the tradeoffs of progressive enhancement (more basic UX functionality, a limited improvement over plain HTML) are worth the benefits for your applications users.

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

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

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

1. Boosting The Entire Contact.app

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

❶ All links and forms will be boosted now!

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

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

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

4.3. Deleting Contacts

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

Listing 1. 29. Plain HTML Form To Delete A Contact

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

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

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

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

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

Listing 1. 30. An htmx Powered Button For Deleting A Contact

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

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

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

4.3.1. Updating The Server Side

We have updated our client-side code, that is our HTML, so it now does "the right thing"

from a hypermedia perspective: we want to delete a contact, and we are issuing a `DELETE` request. But we still have some work to do! Since we updated both the route and the HTTP action we are using, we are going to need to update the server side implementation as well to handle this new HTTP Request.

Here is the original code:

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

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

Secondly, and this is a bit more subtle, we are going to need to change the HTTP Response Code that the handler sends back.

HTTP Response Codes

HTTP Response Codes are numeric values that are embedded in an HTTP response that let the client know what the result of a request was. The most familiar response code for most web developers is `404`, which stands for "Not Found" and is the response code that is returned by web servers when a resource that does not exist is requested.

HTTP breaks response codes up into various categories:

`100-199`	Informational responses that provide information about how the server is processing the response
`200-299`	Successful responses indicating that the request succeeded
`300-399`	Redirection responses indicating that the request should be sent to some other URL
`400-499`	Client error responses indicating that the client made some sort of bad request (e.g. asking for something that didn't exist in the case of `404` errors)
`500-599`	Server error responses indicating that the server encountered an error internally as it attempted to respond to the request

Within each of these categories there are multiple response codes for specific situations. A good example is the `404 Not Found` response code that we already mentioned, which indicates that the requested resource was not fount. This is in contrast with the `403 Forbidden` response code, which is still a "Client Error" response code, but which indicates that the current user is not allowed to view the given resource.

Different response codes will often trigger different browser behaviors, so it is important to understand exactly which one you are returning, especially as you get deeper into creating a Hypermedia Driven Application.

It turns out that, by default, in Flask the `redirect()` method responds with a `302` response code. According to the Mozilla Developer Network (MDN) web docs, this means

that the HTTP method and body of the requests *will be unchanged* when the redirected request is issued.

Since we are issuing a `DELETE` request and being redirected to the `/contacts` path, that would mean that the redirected request would retain the `DELETE` method, and we would issue a `DELETE` request to `/contacts`. Yikes! That looks like a request to delete all the contacts in our system, doesn't it? It wouldn't do that, of course, since we haven't implemented that behavior, but that's still not what we want: we'd like it to simply issue a `GET`, slightly modifying the Post/Redirect/Get behavior we discussed earlier to be Delete/Redirect/Get.

Fortunately for us, there is a response code, `30 See Other`, which will convert the redirected request to a GET. So we want to use this response code in our flask application and, it turns out, this is very easy: there is a second parameter to `redirect()` that takes the numeric response code you wish to send.

Putting It All Together

So we want to make the following changes to our server side code:

- We want to change the HTTP action associated with it to `DELETE`
- We want to remove the ugly `/delete` at the end of the path, since we are now using a proper HTTP action
- We want to be sure to issue a `303 See Other` response code so we properly issue a `GET` after the redirect

Here is our updated code:

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

❶ A slightly different path and method for the handler

❷ The response code is now a 303

Now, when you want to remove a given contact, you can simply issue a `DELETE` to the same URL as you used to access the contact in the first place. A much more natural hypermedia approach to deleting a resource!

4.3.2. Targeting The Right Element

We aren't quite out of the woods yet, however. As you may recall, by default htmx "targets" the element that triggers a request, and will place the HTML returned by the server inside that element. In this case, since the redirect to `/contacts` is going to re-render the entire contact list, we will end up in the unfortunate situation where the entire list ends up *inside* the "Delete Contact" button!

Mis-targeting elements comes up from time to time in htmx and can lead to some pretty funny situations.

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

Listing 1. 31. A fixed htmx Powered Button For Deleting A Contact

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

❶ We have added an explicit target to the button now

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

4.3.3. Updating The Location Bar URL Properly

Well, almost.

If you click on the button you will notice that, despite the redirect, the URL in the location bar is not correct. It still points to `/contacts/{{ contact.id }}`. This is because we haven't told htmx to update the URL: it just issues the `DELETE` request and then updates the DOM with the response.

Boosting will naturally update the location bar for you, mimicing normal anchors and

forms, but here we are building a custom button because we want to issue a `DELETE`, something not possible in plain HTML. We need to let htmx know that we want the resulting URL from this request "pushed" into the location bar. We can achieve this by adding the `hx-push-url` with the value `true`:

Listing 1. 32. Deleting A Contact, Now With Proper Location Information

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

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

Now we are done. We have a button that, all by itself, is able to issue a properly formatted HTTP `DELETE` request to the correct URL, and the UI and location bar are all updated correctly. This was accomplished with three declarative attributes placed directly on the button `hx-delete`, `hx-target` and `hx-push-url`. Not only that, we were able to remove the enclosing form tag as a bonus! Pretty clean!

4.3.4. One Last Thing

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

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

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

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

Listing 1. 33. Confirming Deletion

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

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

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

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

4.3.5. Progressive Enhancement?

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

Progressive Enhancement and a related topic, Accessibility, are hot-button topics in web development. htmx, like most JavaScript libraries, makes it possible to create applications that do not function in the absence of JavaScript. Retaining support for non-JavaScript clients requires additional work and complexity. It is important to determine how important supporting non-JavaScript clients is before you begin using htmx or any other JavaScript framework for improving your web applications.

4.4. Next Steps: Validating Emails

Let’s move on to another improvement in our application: a big part of any web app is validating the data that is submitted to the server side: ensuring emails are correctly formatted and unique, numeric values are valid, dates are acceptable, and so forth.

Currently, our application has a small amount of validation that is done entirely server side andthat displays an error message when an error is detected.

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

Listing 1. 34. Server Side Validation On Contact Update

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

❶ We attempt to save the contact

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

So we attempt to save the contact, and, if the `save()` method returns true, we redirect to the contact's detail page. If the `save()` method does not return true, that indicates that there was a validation error and so, instead of redirecting we re-render the HTML for editing the contact. This gives the user a chance to correct the errors, which are displayed along side the inputs.

Let's tak a look at the HTML for the email input:

Listing 1. 35. Validation Error Messages

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

❶ Display any errors associated with the email field

We have a label for the input, an input of type `text` and then a bit of HTML to display any error messages associated with the email.

Server Side Validation

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

Again, we are not going to go into the details in the interest of staying focused on hypermedia, but whatever server side framework you are using almost certainly has some sort of infrastructure available for validating data and collecting errors to display to the user.

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

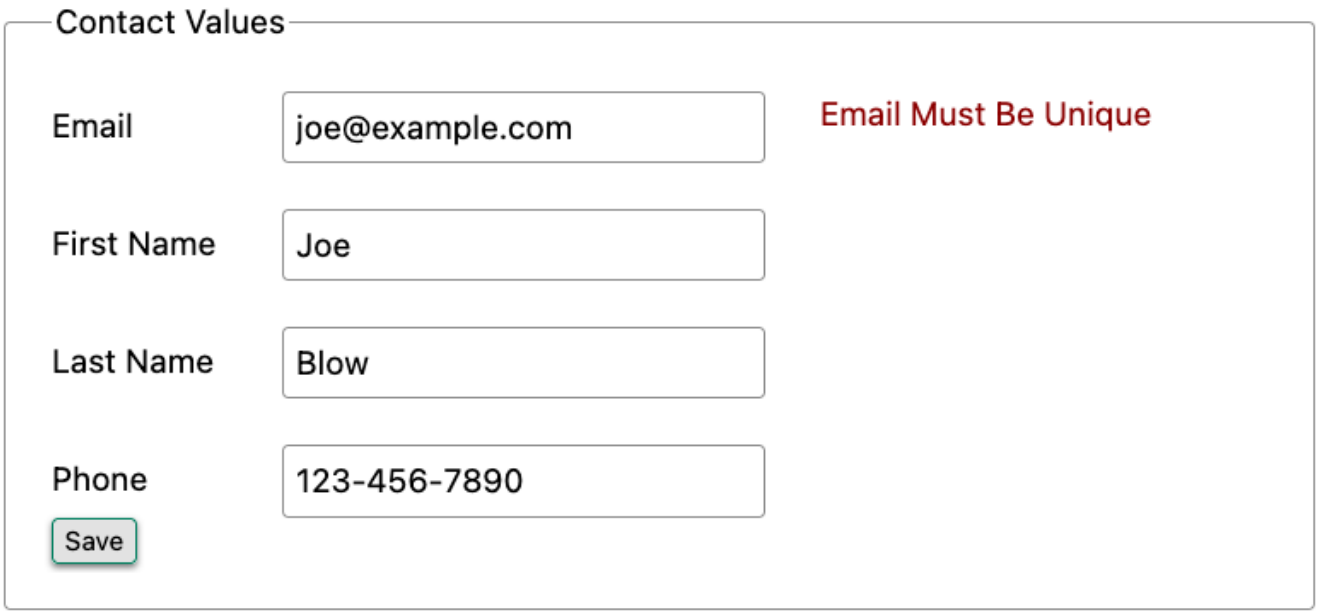

Figure 1. 2. Email Validation Error

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

- First, there is no email format validation: you can enter whatever characters you'd like as an email and, as long as they are unique, the system will allow it

- Second, if a user has entered a duplicate email, they will not find this fact out until they have filled in all the fields because we only check the email's uniqueness when all the data is submitted. This could be quite annoying if the user was accidentally reentering a contact and had to put all the contact information in before being made aware of this fact!

4.4.1. Updating Our Input Type

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

Listing 1. 36. Changing The Input To Type `email`

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

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

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

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

Not bad!

Server Side vs. Client Side Validations

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

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

4.4.2. Inline Validation

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

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

Recall the current HTML for our email input:

Listing 1. 37. The Initial Email Configuration

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

❶ This is the input that we want to have drive an HTTP request to validate the email

❷ This is the span we want to put the error message, if any, into

So we want to add an `hx-get` to this input, which will cause it to issue an HTTP `GET` request to a given URL to validate the email. Then we want to target the error span following the input with any error message returned from the server.

Let's make those changes to our HTML:

Listing 1. 38. Our Updated HTML

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

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

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

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

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

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

Here is the code:

Listing 1. 39. Our Email Validation End-Point

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

❶ Look up the contact by id

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

❸ Validate the contact

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

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

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

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

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

4.4.3. Taking Our User Experience Further

Now, despite the fact that we haven't written a lot of code here, this is a fairly sophisticated

user interface, at least when compared with plain HTML-based applications. However, if you have used more advanced web applications you have probably seen the pattern where an email field (or similar) is validated *as you type*.

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

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

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

Listing 1. 40. Triggering With `keyup` Events

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

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

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

4.4.4. Debouncing Our Validation Requests

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

htmx supports a `delay` modifier for triggers that allows you to debounce a request by adding a delay before the request is sent. If another event of the same kind appears within that interval, htmx will not issue the request and will reset the timer. This is exactly what

we want for this situation: if the user is busy typing in an email we won't interrupt them, but as soon as they pause or leave the field, we'll issue a request.

Let's add a delay of 200 milliseconds to the `keyup` trigger, which is long enough to detect that the user has stopped typing.:

Listing 1. 41. Debouncing the `keyup` Event

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

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

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

4.4.5. Ignoring Non-Mutating Keys

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

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

Listing 1. 42. Only Sending Requests When The Input Value Changes

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

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

Now that's some pretty good-looking code! With a total of three attributes and a simple new server-side end point, we have added a fairly sophisticated user experience to our web application. Even better, any email validation rules we add on the server side will *automatically* just work using this model: because we are using hypermedia as our communication mechanism there is no need to keep a client-side and server-side model in sync with one another.

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

4.5. Another Improvement: Paging

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

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

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

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

now for our simple application.

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

Listing 1. 43. Adding Paging Widgets To Our List of Contacts

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

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

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

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

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

Now lets address the server side implementation.

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

Listing 1. 44. Adding Paging To Our Request Handler

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

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

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

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

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

4.5.1. Click To Load

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

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

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

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

Listing 1. 45. Changing To "Click To Load"

```
        <tbody>
        {% for contact in contacts %}
            <tr>
                <td>{{ contact.first }}</td>
                <td>{{ contact.last }}</td>
                <td>{{ contact.phone }}</td>
                <td>{{ contact.email }}</td>
                <td><a href="/contacts/{{ contact.id }}/edit">Edit</a> <a
href="/contacts/{{ contact.id }}">View</a></td>
            </tr>
        {% endfor %}
        {% if contacts|length == 10 %} ❶
            <tr>
                <td colspan="5" style="text-align: center">
                    <button hx-target="closest tr" ❷
                            hx-swap="outerHTML"   ❸
                            hx-select="tbody > tr" ❹
                            hx-get="/contacts?page={{ page + 1 }}">Load More</button>
                </td>
            </tr>
        {% endif %}
        </tbody>
```

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

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

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

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

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

4.5.2. Infinite Scroll

Another somewhat common pattern for dealing with long lists of things is known as

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

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

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

Listing 1. 46. Changing To "Infinite Scroll"

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

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

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

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

the web was designed!

4.6. Summary

- In this chapter we began improving our Hypermedia-Driven Application (HDA) by using the htmx library
- A simple and quick way to improve the application was to use the `hx-boost` attribute, which "boosts" all links and forms to use AJAX interactions
- Deleting a contact could be updated to use the proper `DELETE` HTTP request, using the `hx-delete` attribute
- Validating the email of a contact as the user entered it was achieved using a combination of `hx-get` and `hx-target`
- Paging was added to the application using standard server-side techniques, then the "Click To Load" and "Infinite Scroll" patterns.

5. Hypermedia In Action

6. Advanced Hypermedia Patterns

This chapter covers:

- Adding the "Active Search" pattern to our application
- Adding the "Lazy Load" pattern to our application
- Implementing inline deletion of contacts from the list view
- Implementing a bulk delete of contacts

6.1. Active Search

In this chapter we will add some more advanced features to our contacts application, all while staying within the hypermedia model. (We will do some client-side scripting in our application later on, but, even when we do add scripting based features, we will keep the network communication model firmly within the hypermedia architecture!)

The first advanced feature we will create is known as the "Active Search" pattern. Active Search is a feature when, as a user types text into a search box, the results of that search are dynamically updated. This pattern was made popular when Google adopted it for search

results, and many applications now implement it.

As you might suspect, we are going to use some of the same techniques we used for dynamically updating emails in the previous chapter, since we are once again going to want to issue requests on the `keyup` event.

Let's look at the current search field in our application once again:

Listing 5. 47. Our Search Form

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

You will recall that we have some server side code that looks for the `q` parameter and, if it is present, searches the contacts for that term.

As it stands right now, the user must hit enter when the search input is focused, or click the "Search" button. Both of these events will trigger a `submit` event on the form, causing it to issue an HTTP `GET` and re-rendering the whole page. Recall that currently, thanks to `hx-boost` the form will still use an AJAX request for this `GET`, but we don't get the nice search-as-you-type behavior we want.

To add active search behavior, we will need to add a few htmx attributes to the search input. We will leave the form as is, so that, in case a user does not have JavaScript enabled, search continues to work. (As a reminder, this is called "progressive enhancement" and this pattern is progressive.) We want to issue an HTTP `GET` request to the same URL that the form does when it is submitted. And we want to do so when a key up occurs, but only after a small delay. We can take the `hx-trigger` attribute directly from our email validation example!

Listing 5. 48. Adding Active Search Behavior

```
<form action="/contacts" method="get" class="tool-bar">
    <label for="search">Search Term</label>
    <input id="search" type="search" name="q" value="{{ request.args.get('q') or ''
}}" ❶
           hx-get="/contacts" ❷
           hx-trigger="search, keyup delay:200ms changed"/> ❸
    <input type="submit" value="Search"/>
</form>
```

❶ We keep everything the same on the input, so it functions the way it always has if JavaScript isn't enabled

❷ We issue a `GET` to the same URL as the form

❸ We use a similar `hx-trigger` specification as we did for the email input validation example

The small change that we made to the `hx-trigger` attribute is we switched out the `change` event for the `search` event. The `search` event is triggered when someone clears the search or hits the enter key. It is a non-standard event, but doesn't hurt to include here. The main functionality of the feature is provided by the second triggering event, the `keyup` which, as with the email example, is delayed to debounce the input requests and avoid issuing too many requests.

What we have is pretty close to what we want, but recall that the default target for an element is itself. As things currently stand, an HTTP `GET` request will be issued to the `/contacts` path, which will, as of now, return an entire HTML document of search results! This whole document will then be inserted into the inner HTML of an input! Well, that's pretty meaningless, and the browser will, sensibly, just ignore htmx's request to do this. So, at this point, when a user types anything into our input, a request will be issued, but it will appear to the user as if nothing has happened.

OK, so to fix this issue, what do we want to target with the update instead? Ideally we'd like to just target the actual results: there is no reason to update the header or search input, and that could cause an annoying flash as focus jumps around.

Fortunately the `hx-target` attribute allows us to do exactly that! Lets use it to target the results body, the `tbody` element in the table of contacts:

Listing 5. 49. Adding Active Search Behavior

```
<form action="/contacts" method="get" class="tool-bar">
    <label for="search">Search Term</label>
    <input id="search" type="search" name="q" value="{{ request.args.get('q') or ''
}}"
           hx-get="/contacts"
           hx-trigger="change, keyup delay:200ms changed"
           hx-target="tbody"/> ❶
    <input type="submit" value="Search"/>
</form>
```

❶ Target the `tbody` tag on the page

Because there is only one `tbody` on the page, we can use the CSS selector `tbody` and htmx will target the first element matching that selector.

Now if you try typing something into the search box, you'll get some action. A request is made and the results are inserted into the document within the tbody. Unfortunately, the results are… the entire document still! So you get all the other stuff, the search box, the application header, etc. and a somewhat humorous double-render.

Now, we could use the same trick we reached for in the "Click To Load" and "Infinite Scroll" features: `hx-select`. Recall that `hx-select` allows us to pick out the part of the response we are interested in using CSS selectors. So we could add this to our input:

Listing 5. 50. Using `hx-select` for Active Search

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or '' }}"
       hx-get="/contacts"
       hx-trigger="change, keyup delay:200ms changed"
       hx-target="tbody"
       hx-select="tbody tr"/> ❶
```

❶ Adding an `hx-select` that picks out the table rows in the `tbody` of the response

6.1.1. Server Side Tricks With htmx

This works fine, but we are not going to use this approach. Here we are letting the server create a full HTML document in response and then, on the client side, filtering it down. This is easy and might be necessary if we don't control the server side or can't easily modify responses. But here we can modify our server responses so, instead of using this

client-side approach, we are going to use this as an opportunity to explore returning different bits of HTML based on the context information that htmx provides with requests.

Let's take a look again at the server side code for our search logic:

Listing 5. 51. Server Side Search

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

❶ This is where the search logic happens

❷ We simply rerender the `index.html` template every time, no matter what

What we want to do on the server side is *conditionally* render only the table rows when we are serving an "Active Search" request. Remember, though, we *also* need to handle "regular" search requests submitted by the form, in case JavaScript is disabled, or the user clicks the "Search" button. In these cases we want the current logic, where we render the entire `index.html` template, to execute.

So we need some way to determine exactly *who* made the request to the `/contact` URL to know what to render. It turns out that htmx helps us out here by including a number of HTTP *Request Headers* when it makes requests. Request Headers are name/value pairs of metadata associated with the request and are a standard, if underutilized, feature of HTTP.

Here are the headers that htmx gives us to work with:

Header	**Description**
HX-Boosted	This will be the string "true" if the request is made via an element using hx-boost
HX-Current-URL	This will be the current URL of the browser

Header	Description
HX-History-Restore-Request	This will be the string "true" if the request is for history restoration after a miss in the local history cache
HX-Prompt	This will contain the user response to an hx-prompt
HX-Request	This value is always "true" for htmx-based requests
HX-Target	This value will be the id of the target element if it exists
HX-Trigger-Name	This value will be the name of the triggered element if it exists
HX-Trigger	This value will be the id of the triggered element if it exists

Looking through this list of headers, the last one stands out: we have an id, `search` on our search input. So the value of the `HX-Trigger` header should be set to `search` when the request is coming from the search input. Perfect!

Let's add some conditional logic to our controller:

Listing 5. 52. Updating Our Server Side Search

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

❶ If the request header `HX-Trigger` is equal to "search", we want to do something different

❷ But what is that something?

OK, we have the conditional logic in place in our controller, but what do we want to do here? Well, we want to do something akin to what we were achieving using `hx-select` previously: we only want to render the *rows* of the table within the table body!

How can we achieve that?

6.1.2. Factoring Your Templates

Here we come to a common pattern in htmx: we want to *factor* our server side templates. This means that we want to break them up a bit so they can be called from multiple contexts. In this situation, we want to break the rows of the results table out to a separate template. We will call this new template `rows.html` and we will include it from the main `index.html` template, as well as render it directly in the controller when we want to respond with only the rows to Active Search requests.

Recall what the table in our `index.html` file currently looks like:

Listing 5. 53. The Contacts Table

```
<table>
    <thead>
    <tr>
        <th>First</th> <th>Last</th> <th>Phone</th> <th>Email</th> <th></th>
    </tr>
    </thead>
    <tbody>
    {% for contact in contacts %} ❷
        <tr>
            <td>{{ contact.first }}</td>
            <td>{{ contact.last }}</td>
            <td>{{ contact.phone }}</td>
            <td>{{ contact.email }}</td>
            <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
                <a href="/contacts/{{ contact.id }}">View</a></td>
        </tr>
    {% endfor %}
    </tbody>
</table>
```

What we want to do is to move that for loop and the rows it creates out so a separate file,

and save that as `row.html`:

Listing 5. 54. Our New `rows.html` file

```
        {% for contact in contacts %} ❷
            <tr>
                <td>{{ contact.first }}</td>
                <td>{{ contact.last }}</td>
                <td>{{ contact.phone }}</td>
                <td>{{ contact.email }}</td>
                <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
                    <a href="/contacts/{{ contact.id }}">View</a></td>
            </tr>
        {% endfor %}
```

We can then include this new file in our table in `index.html` by using the Jinja2 `include` directive:

Listing 5. 55. Including The New File

```
    <table>
        <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Phone</th>
            <th>Email</th>
            <th></th>
        </tr>
        </thead>
        <tbody>
        {% include 'rows.html' %} ❶
        </tbody>
    </table>
```

❶ This directive includes the `rows.html` file, inserting the content from that template into the `index.html` template

So far, so good. The application still works and if we navigate to the `/contacts` page, everything is still rendering properly. But we need to go back and fix up our controller now to take advantage of this new file when we are doing an Active Search. Luckily, the update is simple: we just need to call the `render_template` function with this new file:

Listing 5. 56. Updating Our Server Side Search

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

❶ Render the new template in the case of an active search

Now, when an Active Search request is made, rather than getting an entire HTML document back, we only get a partial bit of HTML, the table rows for the contacts that match the search. These rows are then inserted into the `tbody` on the index page, without any need for an `hx-select` or any other client side processing.

And the old form-based search still works as well, thanks to the fact that we conditionally render the rows only when the `search` input issues the HTTP request.

6.1.3. Updating History

You may have noticed one shortcoming of our Active Search when compared with submitting the form: the form puts the query into the navigation bar as a URL parameter. So if you search for "joe" in the search box, you will end up with a url that looks like this:

`https://example.com/contacts?q=joe`

This features makes it such that you can copy the URL and send it to someone else, and they can simply click on the link to repeat the exact same search. As it stands right now, during Active Search, we do not update the URL.

Let’s fix that by adding the `hx-push-url` attribute:

Listing 5. 57. Updating The URL During Active Search

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or '' }}"
       hx-get="/contacts"
       hx-trigger="change, keyup delay:200ms changed"
       hx-target="tbody"
       hx-push-url="true"/> ❶
```

❶ By adding the `hx-push-url` attribute with the value `true`, htmx will update the URL when it makes a request

That's all it takes and now, as Active Search requests are sent, the URL in the browser is updated to have the query in it, just like when the form is submitted.

Now, you might not *want* this behavior. You might feel it would be confusing to users to see the navigation bar updated and have history entries for every Active Search made, for example. Which is fine! You can simply omit the `hx-push-url` attribute and it will go back ot the behavior you want. htmx tries to be flexible enough that you can achieve the UX you want, while staying largely within the declarative HTML model.

6.1.4. Adding A Request Indicator

A final touch for our Active Search pattern is to add a request indicator to let the user know that a search is in progress. As it stands the user has to know that the active search functionality is doing a request implicitly and, if the search takes a bit, may end up thinking that the feature isn't working. By adding a request indicator we let the user know that the hypermedia application is busy and they can wait (hopefully not too long!) for the request to complete.

htmx provides support for request indicators via the `hx-indicator` attribute. This attribute takes, you guessed it, a CSS selector that points to the indicator for a given element. The indicator can be anything, but it is typically some sort of animated image, such as a gif or svg file, that spins or otherwise communicates visually that "something is happening".

Let's add a spinner next to our input:

Listing 5. 58. Updating The URL During Active Search

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or '' }}"
       hx-get="/contacts"
       hx-trigger="change, keyup delay:200ms changed"
       hx-target="tbody"
       hx-push-url="true"
       hx-indicator="#spinner"/> ❶
<img id="spinner" class="htmx-indicator" src="/static/img/spinning-circles.svg"
alt="Request In Flight..."/> ❷
```

❶ The `hx-indicator` attribute points to the indicator image after the input

❷ The indicator is a spinning circle svg file, and has the `htmx-indicator` class on it

We have added the spinner right after the input. This visually co-locates the request indicator with the element making the request, and makes it easy for a user to see that something is in fact happening.

Note that the indicator `img` tag has the `htmx-indicator` class on it. This is a CSS class automatically injected by htmx that defaults the element to an `opacity` of 0. When an htmx request is triggered that points to this indicator, another class, `htmx-request` is added to the indicator which transitions its opacity to 1. So you can use just about anything as an indicator and it will be hidden by default, and will be shown when a request is in flight. This is all done via standard CSS classes, allowing you to control the transitions and even the mechanism by which the indicator is show (e.g. you might use `display` rather than `opacity`). htmx is flexible in this regard.

Request indicators are an important UX aspect of any distributed application. It is unfortunate that browsers have de-emphasized their native request indicators over time, and it is doubly unfortunate that request indicators are not part of the JavaScript ajax APIs.

Be sure not to neglect this important aspect of your application! Even though requests might seem instant when you are working on your application locally, in the real world