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 "□περ-" 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:

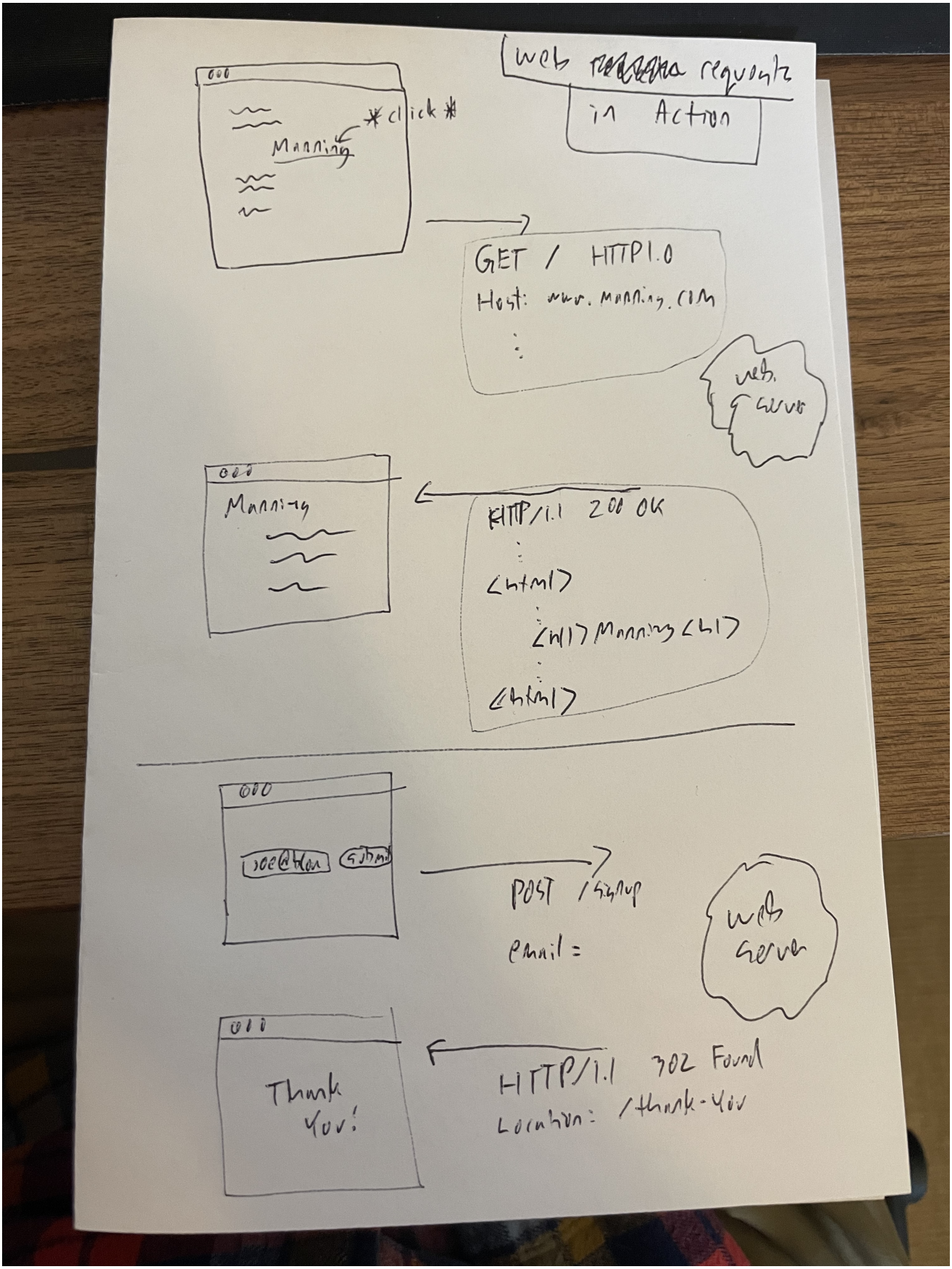

Figure 1. 1. HTTP Requests In Action

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') ①
                 .then(response => response.json()) ②
                 .then(data => updateTable(data))"> ③
    Fetch Contacts
</button>
```

① Issue the request

② Convert the response to a JavaScript object

③ 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

```
{ ①
  "id": 42, ②
  "email" : "json-example@example.org" ③
}
```

① The start of a JSON object

② A property, in this case with the name `id` and the value `42`

③ 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"> ①
    Fetch Contacts
</button>
```

① 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("/") ①
def index(): ②
    return redirect("/contacts") ③
```

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

② The next method is the handler for that route

③ Redirect the request to a new path

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

The List & Search Template

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

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

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

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

```
{% extends 'layout.html' %} ①

{% block content %} ②

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

① Set the layout template for this template

② Delimit the content to be inserted into the layout

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

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

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

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

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

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

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

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

① We output some headers for our table

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

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

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

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

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

- A link to the "Edit" page for the contact, located at `/contacts/{{ contact.id }}/edit` (e.g. For the contact with id 42, the edit link will point to `/contacts/42/edit`)
- A 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> ①
    </p>

{% endblock %} ②
```

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

② The closing element of the `content` block

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

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']) ①
def contacts_new_get():
    return render_template("new.html", contact=Contact()) ②
```

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

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

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

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

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

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

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

② A label for the first form input

③ the first form input, of type email

④ Any error messages associated with this field

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

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

Next we have inputs for the other fields for contacts:

```
        <p>
            <label for="first_name">First Name</label>
            <input name="first_name" id="first_name" type="text" placeholder="First
Name" value="{{ contact.first or '' }}">
            <span class="error">{{ contact.errors['first'] }}</span>
        </p>
        <p>
            <label for="last_name">Last Name</label>
            <input name="last_name" id="last_name" type="text" placeholder="Last Name"
value="{{ contact.last or '' }}">
            <span class="error">{{ contact.errors['last'] }}</span>
        </p>
        <p>
            <label for="phone">Phone</label>
            <input name="phone" id="phone" type="text" placeholder="Phone" value="{{
contact.phone or '' }}">
            <span class="error">{{ contact.errors['phone'] }}</span>
        </p>
```

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

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

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

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

Handling The Post

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

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

Here is our controller code:

```
@app.route("/contacts/new", methods=['POST'])
def contacts_new():
    c = Contact(None, request.form['first_name'], request.form['last_name'],
request.form['phone'],
                request.form['email']) ①
    if c.save(): ②
        flash("Created New Contact!")
        return redirect("/contacts") ③
    else:
        return render_template("new.html", contact=c) ④
```

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

② We try to save it

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

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

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

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

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

If we are able to save the contact (that is, there were no validation errors), we create a *flash* message indicating success and redirect the browser back to the list page. A flash is a common feature in web frameworks that allows you to store a message that will be available on the *next* request, typically in a cookie or in a session store.

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

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

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

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

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

② The handler takes the value of this path parameters

③ Look up the corresponding contact

④ Render the `show.html` template

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

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

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

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

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

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

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

```
              <p>
                  <label for="first_name">First Name</label>
                  <input name="first_name" id="first_name" type="text"
placeholder="First Name"
                         value="{{ contact.first }}">
                  <span class="error">{{ contact.errors['first'] }}</span>
              </p>
              <p>
                  <label for="last_name">Last Name</label>
                  <input name="last_name" id="last_name" type="text" placeholder="Last
Name"
                         value="{{ contact.last }}">
                  <span class="error">{{ contact.errors['last'] }}</span>
              </p>
              <p>
                  <label for="phone">Phone</label>
                  <input name="phone" id="phone" type="text" placeholder="Phone"
value="{{ contact.phone }}">
                  <span class="error">{{ contact.errors['phone'] }}</span>
              </p>
            <button>Save</button>
        </fieldset>
    </form>
```

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

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

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

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

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

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

Factoring Your Applications

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

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

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

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

Handling The Post

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

Here is the definition:

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

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

② Look the contact up by id

③ update the contact with the new information from the form

④ Attempt to save it

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

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

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

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

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"]) ①
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete() ②
    flash("Deleted Contact!")
    return redirect("/contacts") ③
```

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

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

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

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

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

No need for a template in this case!

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 via HTTP. It is 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: make a change to a java object and the web page magically updates.

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 RESTful system.

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 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. 6. 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
- The, when a user clicks on it…
- The browser will issue an HTTP `GET` to `https://www.manning.com` and then…
- The broswer 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 hypermedia mechanic of HTML.

There is no rule saying that hypermedia can *only* work this way, after all!

The first thing to consider 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 is our first opportunity to expand the expressiveness of HTML: we can allow *any* element to issue a request to the server.

Next, let's consider the event that triggers the request to the server on our link, a click. What's so special about clicking (in the case of anchors) or submitting (in the case of forms)? Those are just one 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?

Here is 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 the request.

The next limitation to consider is the technical problem we noted earlier: 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 to get at the other three: `DELETE`, `PUT` and `PATCH`.

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 only have access to two of them is a severe limitation.

This 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`.

Finally, consider the last aspect of a hyperlink, where it replaces the *entire* screen when a user clicks on it. This is what makes for a poor user experience, with flashes of unstyled content, loss of scroll state and so forth. But there is no rule saying that hypermedia exchanges *must* replace the entire document.

Here is our forth 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 issues discussed above and attempts to incrementally address each one, without introducing a significant amount of additional conceptual infrastructure for web developers.

This pure-HTML extension approach is not without tradeoffs: by staying so close to HTML, htmx does not give you a lot of infrastructure that many developers might feel should be there "by default". A good example is the concept of modals. 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.

However, htmx 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 integrate htmx into it if you want to use modals within an htmx-based application.

So this is the design tradeoff that htmx makes: it retains the conceptual purity of a straight extension of HTML in exchange for some of the "batteries included" features found in other front end libraries.

It’s worth nothing that htmx *can* be used to effectively implement a 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. We will take a look at inline editing in the next chapter.

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. 7. 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. 8. A Simple htmx-Powered Button

```
<button hx-get="/contacts"> ①
  Get The Contacts
</button>
```

① 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.

Now we get to perhaps the most important thing to understand about htmx: it expects the response to this AJAX request *to be HTML*, not JSON! htmx is an extension of HTML and, just as the response to an anchor tag click or form submission is typically HTML, htmx expects the server to respond with a hypermedia, namely with HTML.

htmx vs. "normal" responses

We have established that htmx is expecting 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. 9. 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 swapped. 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. 10. A Simple htmx-Powered Button

```
<div id="main"> ①

  <button hx-get="/contacts" hx-target="#main"> ②
    Get The Contacts
  </button>

</div>
```

① A `div` element that wraps the button

② 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. 11. 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. The last two values are specific to htmx, but are fairly obvious to 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. 12. Replacing the Entire div

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML"> ①
    Get The Contacts
  </button>

</div>
```

① 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. 13. 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. 14. A Terrible Idea, But It Demonstrates The Concept!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="mouseenter"> ①
    Get The Contacts
  </button>

</div>
```

① Issue a request… on mouseenter?

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. 15. A Start

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup"> ①
    Get The Contacts
  </button>

</div>
```

① 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. 16. Better!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup[ctrlKey && key == 'l']"> ①
    Get The Contacts
  </button>

</div>
```

① 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. 17. Better!

```
<div id="main">

  <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-trigger="click,
keyup[ctrlKey && key == 'L'] from:body">①
    Get The Contacts
  </button>

</div>
```

① 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. 18. A Simple htmx-Powered Button

```
<div id="main">

  <form> ①
      <label for="search">Search Contacts:</label>
      <input id="search" name="q" type="search" placeholder="Search Contacts"> ②
      <button hx-post="/contacts" hx-target="#main"> ③
        Search The Contacts
      </button>
  </form>

</div>
```

① The form tag encloses the button, thereby including all values within it in the button request

② A new input that users will be able to enter search text into

③ 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. 19. 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">①
    Search The Contacts
  </button>

</div>
```

① 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. 20. A Simple htmx-Powered Button

```
<button hx-get="/contacts" hx-vals='{"state":"MT"}'> ①
  Get The Contacts In Montana
</button>
```

① 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. 21. 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. 22. Our trusty button, now with history!

```
<button hx-get="/contacts" hx-target="#main" hx-push-url="true">①
  Get The Contacts
</button>
```

① `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!

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