Hypermedia In Action

1. Reintroducing Hypermedia

This chapter covers

- An introduction 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?

> Hypertexts: new forms of writing, appearing on computer screens, that will branch or perform at the reader's command. A hypertext is a non-sequential piece of writing; only the computer display makes it practical.
>
> — Ted Nelson, https://archive.org/details/SelectedPapers1977/page/n7/mode/2up

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. The prefix "hyper-" derives from the Greek prefix "ὑπερ-" which means "beyond" or "over", indicating that hypermedia *goes beyond* normal, passively consumed media. Things like hyperlinks are *hypermedia controls*, that is, elements that allow the reader to control and interact with the hypermedia.

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.

Here is what a user interaction with an anchor tag looks like in visual form:

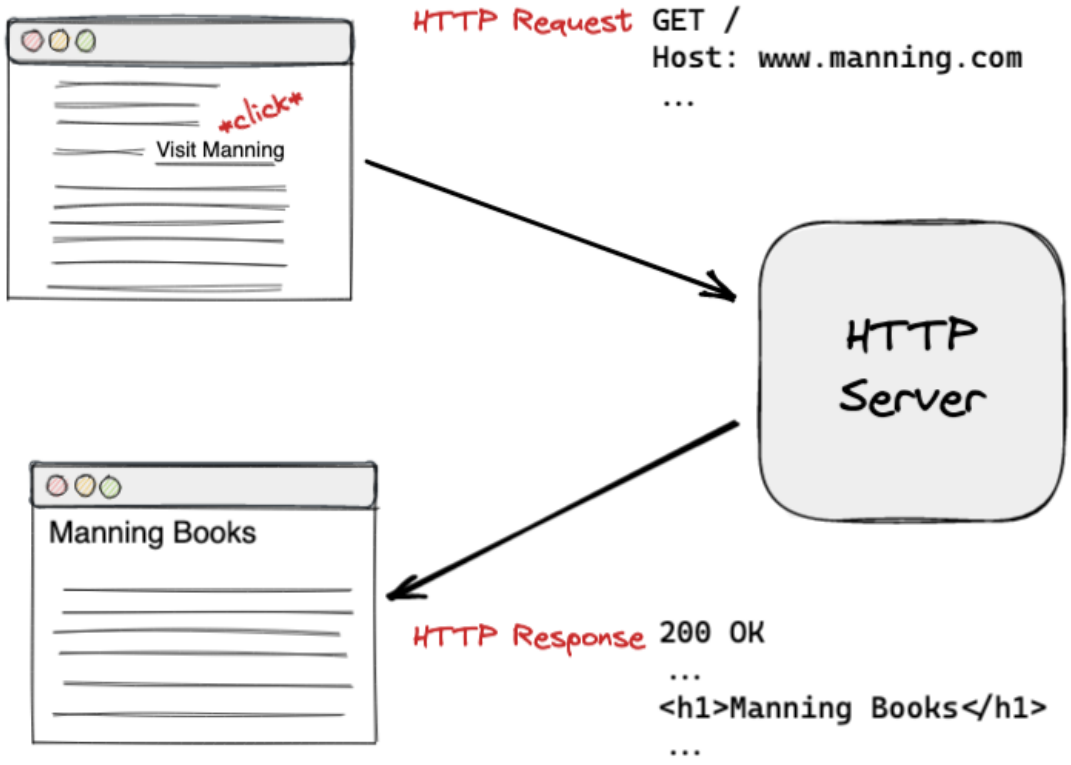


Figure 1. 1. An HTTP GET In Action

You can see that the link initiates an HTTP `GET` request when it is clicked, which has some includes some data on what, exactly, the browser wants from the server, and the server responds with more hypermedia content, that is, with HTML.

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 what a form looks like in action:

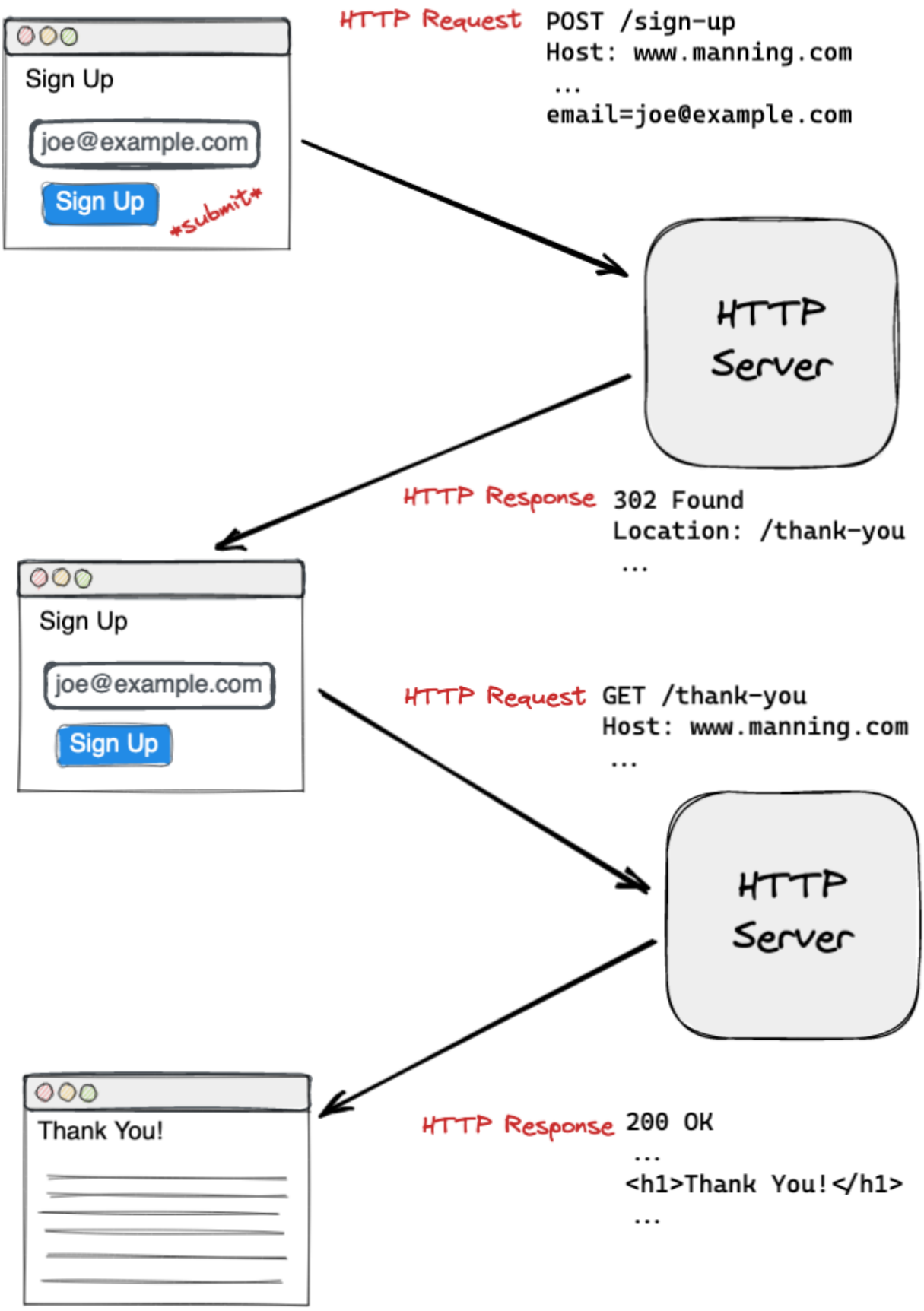


Figure 1. 2. An HTTP POST In Action

In this sequence of requests and responses, the form issues an HTTP `POST` to the server, gets a redirect response, and then issues an HTTP `GET` request to the server for the final content. This is a classic example of the Post/Redirect/Get pattern, which we will discuss a bit later in the book.

I want you to keep these diagrams in your head as a mental model while you read the rest of this the book. This is the pattern that we are going to build on as we go along: exchanging *hypermedia* with a server in response to user events.

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

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

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

1.1.2. So What Isn't Hypermedia?

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

Listing 1. 3. Javascript

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

❶ Issue the request

❷ Convert the response to a JavaScript object

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

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

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

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

Listing 1. 4. JSON

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

❶ The start of a JSON object

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

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

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

What is important to understand about this server interaction is that it is *not* using hypermedia. The JSON API being used here does not return a hypermedia-style response. There are no *hyperlinks* or other hypermedia-style controls in it. This is, rather, a *Data API*. It is returning simple, Plain Old JSON(POJ) formatted data. We say "POJ" here because,

when XML was being used rather than JSON, the term for an API like this was "Plain Old XML", or POX. The term POX was disparaging at the time, sometimes called "The Swamp of POX", but, today, the POJ style of HTTP API is ubiquitous.

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

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

Today, of course, the vast majority of Simple Page Applications adopt far more sophisticated frameworks for managing their user interface than this simple example shows. Libraries like React, Angular, Vue.js, etc. are all popular ways to manage far more complex user interactions than our little demo. With these more complex frameworks you will typically work with a much more elaborate client-side model (that is, JavaScript objects stored locally in the browser's memory that represent the "model" or "domain" of your application.) You then update these JavaScript objects and allow the framework to "react" to those changes via infrastructure baked into the framework itself, which will have the effect of updating the user interface. (This approach, while not new, popularized the term "Reactive Programming" in web development.)

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 popular (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 is extremely tolerant of content and API changes (in fact, it thrives on them!)
- It leverages tried and true features of web browsers, such as caching

As someone interested in web development, these advantages should sound appealing to you. The first two, 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!)
- JSON API churn, that is, the constant changes made JSON APIs to support application needs, has become a huge pain for many application developers

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: the expressiveness of hypermedia (and HTML in particular) hasn't changed much *since the late 1990s*, in terms of user interaction with a server. Sure, many 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 as hypermedia controls, and those hypermedia controls can still only issue `GET` and `POST` requests, despite the fact that other types of HTTP requests have existed almost from the start!

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 *as a hypermedia* 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?

There have been heroic efforts to continue to advance hypermedia outside of HTML, efforts like HyTime, VoiceXML, and HAL. But HTML, the most widely used hypermedia in the world, stopped making progress as a hypermedia, and the web development world moved on, solving the interactivity problem using other tools.

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, if it is 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 technical approach 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 architectural 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!

1.2.3. Hypermedia Driven Applications

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

We rather like to emphasize the *hypermedia* aspect of both the older MPA approach and the

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

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

It might look something like this:

Listing 1. 5. an htmx implementation

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

❶ issues a `GET` request to `/contacts`, replacing the `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, as we walk through building a Hypermedia Driven Application in this book, the differences between the two approaches will become more and more apparent.

This example may seem somewhat cute: this is a simple and a contrived JavaScript example that no one would ever write in production code. And this is a neat demo of a small library that perhaps makes HTML a bit more expressive, sure.

But maybe the htmx example doesn't look very convincing yet. Can this simple concept really scale up to the needs of sophisticated, modern web applications?

In fact, for many applications, it can: just as the original web scaled up shockingly well via hypermedia, this approach can go far further than a first glance suggests. Despite its simplicity (or, perhaps, because of it) you will be surprised just how far we can get in creating dynamic and interactive 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 the interaction with the server is driven by hypertext.

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.

1.6. Giving Hypermedia A Fair Hearing

In today’s web development world, sadly, hypermedia is often an afterthought. Many web developers don’t have a good understanding of the underlying concepts, regarding it as an antiquated approach to building web applications. This is unfortunate for many reasons, there are two that really stand out.

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 "hypermedia library" form, 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 and make the most of your complexity budget.)

In this book we hope to give hypermedia a fair hearing, and show you just how much you can accomplish with this novel and ubiquitous technology.

1.7. Summary

- Hypermedia is a unique technology that extends the concept of "media" to include non-linear user interaction, and has shown itself to be a powerful technology for building large-scale, distributed applications
- Using Data APIs, that is, APIs that do not utilize a hypermedia like HTML, is very common in today's web development world, has significantly different characteristics 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 and search for Contact.App

To begin our journey into Hypermedia Driven Applications, we are going to create a simple contact management web application called 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 building the application, over the coming chapters, 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 that paradigm.

2.1. 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:

- We will use Python as our programming language (https://www.python.org/)
- We will use Flask as our web framework, allowing us to connect HTTP requests to python logic (https://palletsprojects.com/p/flask/)
- We will use Jinja2 for our server-side templating language, allowing us to render HTML responses using a familiar and intuitive syntax (https://palletsprojects.com/p/jinja/)

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 as of this writing, according to the TIOBE index (https://www.tiobe.com/tiobe-index/), a popular measure of programming language popularity. 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.

I picked flask as the web framework because it is very lightweight 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, where lots of functionality is available out of the box.

I understand that perspective, but, for teaching purposes, I feel that an un-opinionated 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.

With Jinja2, we will be rendering our HTML *on the server side*. This is the traditional approach to building web applications, but, with the rise of SPAs, is not as widely known a technique as it once was. Today, as people are rediscovering this approach to building web applications, the term "Server Side Rendering" or SSR is emerging as the way that people talk about this style of templating, in contrast with "Client Side Rendering", that is, rendering templates in the browser, as is common in SPA libraries.

In general, in Contact.app, we will intentionally keep things as simple as possible (sometimes sacrificing other design principles to do so!) to maximize the teaching value of our code: it won't be perfectly factored code, and it certainly won't be the most beautiful web application, but it will be easy enough to follow for readers, and easy to translate into your preferred language and web stack.

2.2. 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 like a better world to us, and that's HOWL, hypermedia on whatever you'd like!

2.3. Python

Since this book is intended to teach how hypermedia can be used effectively, we aren't going to do deep dives into the various technologies we use *around* that hypermedia. This has some obvious drawbacks: if you aren't comfortable with python, for example, some of the example python code in the book may seem confusing or mysterious.

If you feel like you need a quick introduction to the language, we heartily recommend "The Quick Python Book" by Manning Publishing. It gives you a clear and brisk introduction to the language, and will put you in a good position to understand the relatively simple python code used in this book.

That being said, I think most web developers, even developers who are unfamiliar with python, should be able to follow along in the code. Remember, I hadn't written very much python before writing this book, and I got the hang of it pretty quick!

2.4. Introducing Flask: Our First Route

Flask is a very simple but flexible web framework for Python. Just like this book isn't a Python book, it isn't a Flask book either, so we will not go into too much detail about it. However, unlike Python, which is similar in many ways to other programming languages, Flask might be a bit different than the web frameworks you are familiar with, so it is necessary to do a bit more of an introduction to prepare you for the coming chapters. Thankfully, 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. I will use the term "handler" to refer to the functions associated with a route.

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, a simple "Hello World" route. 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 a python function (i.e. handler) to be executed by the server when a request to that path is made.

Here is the code:

Listing 1. 6. A Simple "Hello World" Route

```
@app.route("/") ❶
def index(): ❷
    return "Hello World!" ❸
```

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

❷ The next method is the handler for that route

❸ Returns the string "Hello World!" to the client

The Flask pattern for doing this is to use the `route()` method on the Flask application object, and pass in the path you wish the 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()`. In flask, the function immediately following a route definition is the "handler" for that route, and will be executed when an HTTP request to the given path is made. (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 handler in this case is dead simple, it just returns a string, "Hello World!", to the client. This isn’t even hypermedia yet, but, nonetheless, the browser renders it fine:

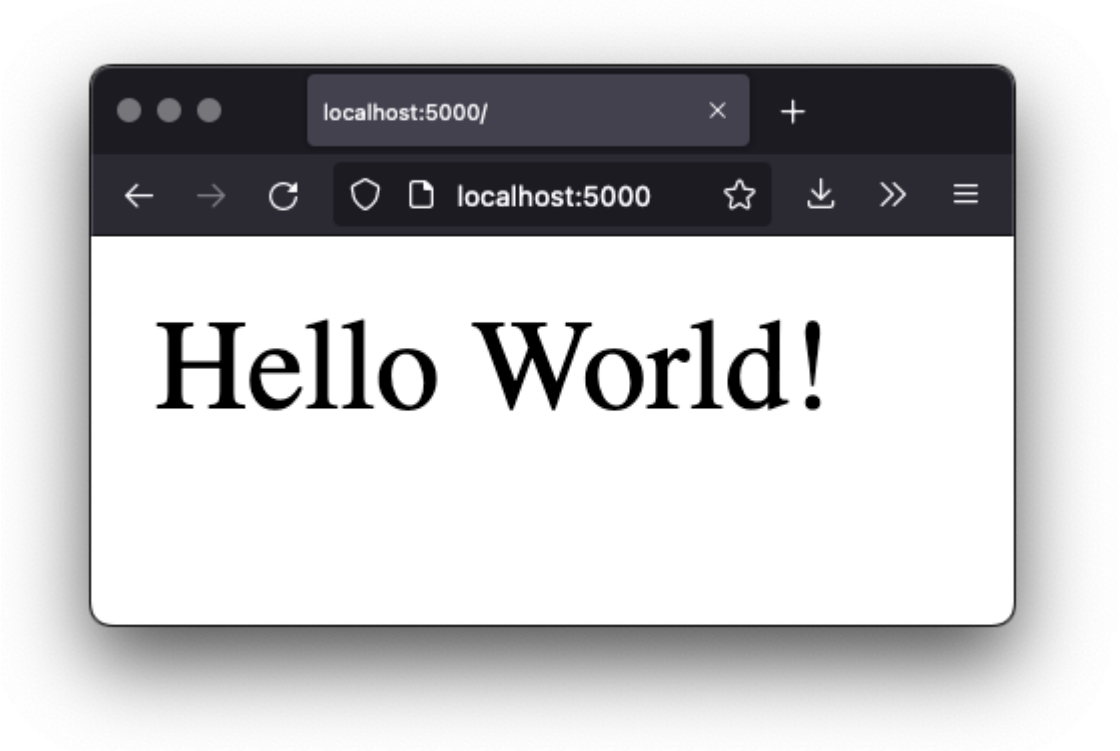


Figure 1. 3. Hello Flask!

For Contact.app, rather than rendering "Hello World!" at the root path, we are going to do something a little fancy: we are going to redirect to another path, the `/contacts` path. Redirects are a feature of HTTP that allow you to, well, redirect a client to another location in an HTTP response. Redirecting to the `/contacts` path is a bit more consistent with notion of resources with REST. It's a judgement call on our part, but we are going to go with it.

To change our "Hello World" route to a redirect, we only need to change one line of code:

Listing 1. 7. Changing "Hello World" to a Redirect

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

❶ Update to a call to `redirect()`

Now the `index()` function simply returns the result of calling a `redirect()` function with the path we with to redirect to, in this case `/contacts`, passed in 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!

2.5. 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. The source code of the application is available at https://github.com/bigskysoftware/contact-app.

2.5.1. Showing A Searchable List Of Contacts

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

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

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

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

Here is the code:

Listing 1. 8. Server Side Search

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

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

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

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

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

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

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

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

Note that we are not going to dig into the code in the `Contact` class. The implementation of the `Contact` class is not relevant to hypermedia, we will ask you to simply accept that it is a "normal" domain model class, and the methods on it act in the "normal" manner. We

will treat `Contact` as a *resource* and will provide hypermedia representations of that resource to clients, in the form of HTML generated via server side templates.

The List & Search Template

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

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

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

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

Listing 1. 9. Start of index.html

```
{% extends 'layout.html' %} ❶

{% block content %} ❷

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

❶ Set the layout template for this template

❷ Delimit the content to be inserted into the layout

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

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

The first line of code references a base template, `layout.html`, with the `extends`

directive. This layout template provides the layout for the page (again, sometimes called "the chrome"): it imports any necessary CSS and scripts, includes a `<head>` element, and so forth.

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

Next we have our first bit of actual HTML, rather than just Jinja directives. We have a simple HTML form that allows you to search contacts by issuing a `GET` request to the `/contacts` path. The form itself contains a lable and an input with the name "q". This input's value will be submitted with the `GET` request to the `/contacts` path.

Note that the value of this input is set to the Jinja expression `{{ request.args.get('q') or '' }}`. This expression is evaluated by Jinja and will insert the request value of "q" as the input's value, if it exists. This will "preserve" the search value when a user does a search, so the text box contains the value that they typed in initially, and makes for a slightly nicer user experience.

Finally, we have a submit-type input, which, when clicked on, will trigger the form to issue an HTTP request.

This search UI forms the top of our contact page, and it is followed a table of all the contacts that are are stored on the server. (Or, if there is a search term, the contacts that match that search term. More on that in a bit.)

Here is what the template code looks like:

Listing 1. 10. The Contacts Table

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

❶ We output some headers for our table

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

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

❹ An "operations" column, with links to edit or view the contact details

Here we are into the core 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`)

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

Listing 1. 11. The Add Contact Link

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

{% endblock %} ❷
```

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

❷ The closing element of the `content` block

And that's our template! Using this server side template, in combination with our handler method, we can respond with an HTML *representation* of all the contacts requested. So far, so hypermedia!

Here is what the template looks like, rendered with a bit of contact data:

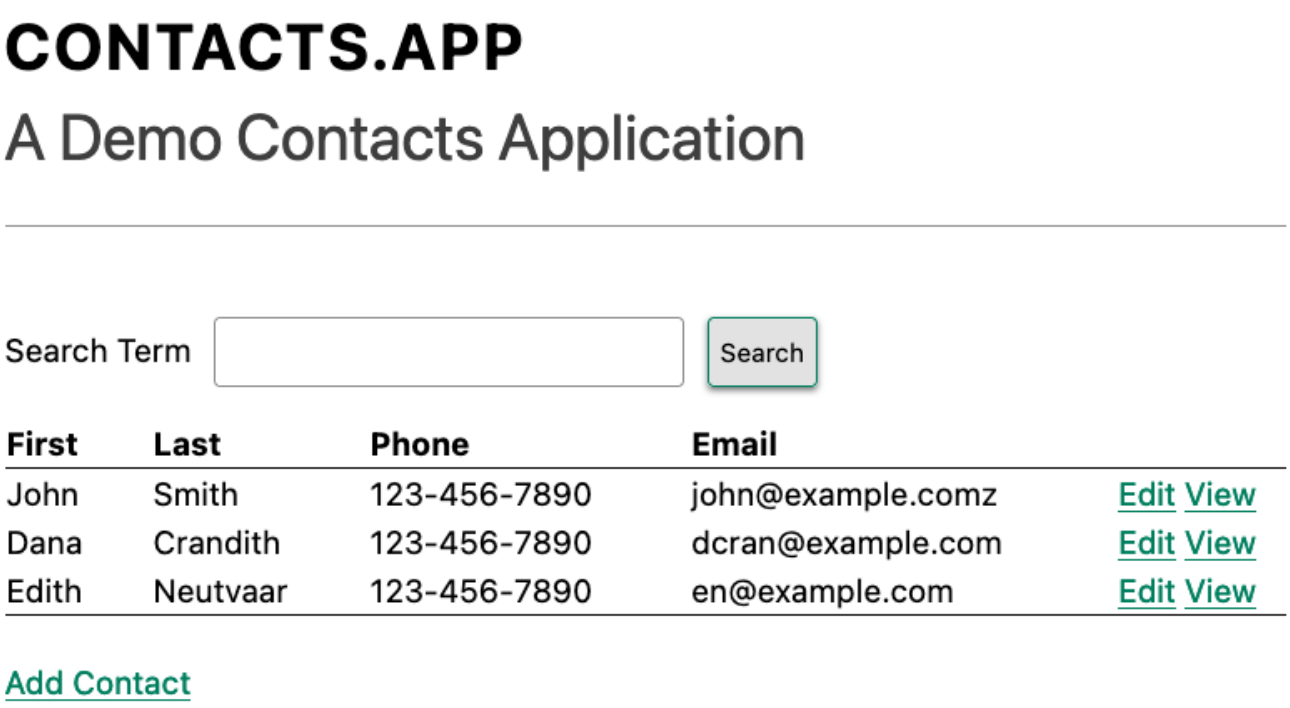


Figure 1. 4. Contact.App

It won't win any design awards at this poitn, but 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.

As simple as it is, this is a very REST-ful application!

2.5.2. 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:

Listing 1. 12. The New Contact GET Route

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

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

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

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

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

If you are familiar with HTML you are probably expecting a form element here, and you

will not be disappointed. We are going to use the standard form element for collecting contact information and submitting it to the server.

Here is what our HTML looks like:

Listing 1. 13. A New Contact Form

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

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

❷ A label for the first form input

❸ the first form input, of type email

❹ Any error messages associated with this field

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

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

Next we have inputs for the other fields for contacts:

Listing 1. 14. Inputs And Labels For The New Contact Form

```
        <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:

Listing 1. 15. The Submit Button For The New Contact Form

```
        <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 to `/contacts/new`

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:

Listing 1. 16. The New Contact Controller Code

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

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

❷ We try to save it

❸ On success, "flash" a success message and redirect back to the `/contacts` page

❹ On failure, 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!

2.5.3. 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`

Path Design In Your HDA

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. I feel it is more important to understand the overarching idea of resources and the hypermedia representations of them, rather than hairsplitting about path layouts: just pick a reasonable 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.

Here is what the code looks like, just a few lines of simple Python:

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

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

❷ The handler takes the value of this path parameters

❸ Look up the corresponding contact

❹ Render the `show.html` template

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

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

2.5.4. 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:

Listing 1. 17. The Contact Details 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!

2.5.5. 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:

Listing 1. 18. The Edit Contact Controller Code

```
@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:

Listing 1. 19. The Edit Contact Form Start

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

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

❷ As with the `new.html` page, the input is tied to the contact's email

This HTML is 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 we want to update. It's worth nothing here that, rather than `POST`, I would prefer to use a `PUT` or `PATCH`, but those are not available in plain HTML.

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.

Listing 1. 20. The Edit Contact Form Body

```
            <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. As with being able to use a `PUT` to update a contact, it sure would be nice if we could issue a `DELETE` request to delete one, but unfortunately that also isn't possible in plain HTML!

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

Listing 1. 21. The Edit Contact Form Footer

```
<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 our application is 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.

With Hypermedia Driven 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 to `/contacts/<contact_id>`

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 our Python code:

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

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

❷ Look the contact up by id

❸ update the contact with the new information from the form

❹ Attempt to save it

❺ On success, flash a success message and redirect to the contacts detail page

❻ On failure, 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!

2.5.6. 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

Listing 1. 22. The Delete Contact Controller Code

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

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

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

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

The handler code is very simple since we don't need to do any validation or conditional logic: we simply look up the contact the same way we have been doing in our other handlers and invoke the `delete()` method on it, then redirect back to the list of contacts with a success flash message.

No need for a template in this case!

2.5.7. 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.

2.6. Summary

- A Hypermedia Driven Application is an application that primarily relies on exchanging hypermedia with a server for achieving interactivity
- Remember that Web 1.0 applications are, by definition, Hypermedia Driven Applications!
- Flask is a simple Python library for connecting routes to server-side logic, or handlers, and provides a good foundation for building simple Hypermedia Driven Applications
- Jinja2 is the default templating library used for Server Side Rendering in Flask, it is a fairly standard templating library
- Combining Flask and Jinja templates allowed us to implement a basic CRUD-style Web

1.0 application for managing contacts in a basic web 1.0-style application with relatively little effort

- This web 1.0-style application works fine, but it feels a bit clunky and old fashioned. We'll look into how to fix that problem, while still using hypermedia, 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

- Some shortcomings of "plain" HTML
- How htmx addresses these opportunties to improve the expressiveness of HTML
- How to issue various HTTP requests with htmx
- History and back button support in htmx

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

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

Two obvious problems that jump out are:

- From a user experience perspective: there is a noticeable refresh when you move between pages of the application, of when you create, update or delete a contact. This is because every user interaction (link click or form submission) requires a full page refresh, with a whole new HTML document to process after each action.

- From a technical perspective, all the updates are done with the `POST` HTTP action. This is despite the fact that more logical actions HTTP request types like `PUT` and `DELETE` exist and would make far more sense for some operations. After all, if we wanted to delete a resource, wouldn't it make more sense to use an HTTP `DELETE` request to do so? Somewhat ironically, since we are using pure HTML, we are unable to access the full expressive power of HTTP, which was designed specifically *for* HTML!

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, by mutating the Document Object Model (DOM). The DOM is the internal model that a browser builds up when it processes HTML, forming a tree of "nodes" for the tags and other content in the HTML. The DOM provides a programmatic JavaScript API that allows you to update the nodes in a page directly, without the use of hypermedia. Using this API, JavaScript code can insert new content, or remove or update existing content, entirely outside the normal browser request mechanism.

There are a few of different styles of SPA, but, as we discussed in Chapter 1, the most common approach today is to tie the DOM to a JavaScript model and then let an SPA framework like react (https://reactjs.org/) or vue (https://vuejs.org/) *reactively* update the DOM when the JavaScript model is updated: you make a change to a JavaScript object and the web page magically updates its state to reflect the change in the model.

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

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

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

distributed networking architecture that is flexible and resilient.

3.1. A Close Look At A Hyperlink

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

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

Listing 1. 23. A Simple Hyperlink, Again

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

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

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

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

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

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

to issue requests?

This presents our first opportunity to expand the expressiveness of HTML:

IMPORTANT	**Opportunity 1** HTML could be extended to allow *any* element to issue a request to the server

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

This gives us our second opportunity to expand the expressiveness of HTML:

IMPORTANT	**Opportunity 2** HTML could be extended to allow *any* event, not just a click, as in the case of our hyperlinks, to trigger HTTP requests

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

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

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

complete replacement of the entity

- `DELETE` deletes the given resource

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

So here is our third opportunity to expand the expressiveness of HTML:

IMPORTANT	**Opportunity 3** HTML could be extended so that it could access these missing three HTTP actions, `PUT`, `PATCH` and `DELETE`.

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

This gives us our forth, final and perhaps most important opportunity to generalize HTML:

IMPORTANT	**Opportunity 4** HTML could be extended to allow the responses to requests to replace elements *within* the current document, rather than requiring that they replace the *entire* document

This is actually a very old concept in hypermedia. Ted Nelson, in his 1980 book "Literary Machines" coined the term *transclusion* to capture this idea: the inclusion of content into an existing document via a hypermedia reference. If HTML supported this style of "dynamic transclusion", then Hypermedia Driven Applications could 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. We wouldn’t be changing the fundamentals of the HTML and the HTTP requests that we use to build our application. Rather, these four generalizations of existing functionality already found within HTML would simply let us

accomplish *more* using 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 shortcomings of HTML as a hypermedia: it has full access to the DOM and is nearly universally available!

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, there are other excellent libraries like Unpoly (https://unpoly.com) and Hotwire (https://hotwire.dev). But htmx is perhaps the purest of these libraries in the pursuit of extending HTML as a hypermedia.

It focuses on the four opportunities for improving HTML that we discussed above and attempts to incrementally address each one, without introducing a significant amount of additional conceptual infrastructure for web developers beyond what is necessary to address those opportunities.

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. 24. 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. 25. A Simple htmx-Powered Button

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

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

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

3.3.1. It's All Just HTML!

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

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

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

htmx vs. "plain" HTML responses

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

In htmx-powered interactions we are typically not replacing the entire document. Rather we are using "transclusion" to include content *within* an existing document. Because of this, it is often not necessary or desirable 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.

Let's consider what a simple *partial* HTML response to the "Get Contacts" button might be when it is clicked.

It might look something like this:

Listing 1. 26. A partial HTML Response to an htmx Request

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

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

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

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

3.4. Targeting Other Elements

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

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

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

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

Listing 1. 27. A Simple htmx-Powered Button

```
<div id="main"> ❶

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

</div>
```

❶ A `div` element that wraps the button

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

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

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

It would look something like this:

Listing 1. 28. 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. Transclusion! And this 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 server response *into* the div, as child elements. Perhaps, for whatever reasons, we wish to *replace* the entire div with the response. Seems like a reasonable behavior to make configurable.

As luck would have it, htmx provides another attribute, `hx-swap`, that allows you to specify exactly *how* the content should be swapped into the DOM. (Are you beginning to sense a pattern here?)

The `hx-swap` attribute supports the following values:

- `innerHTML` - The default, replace the inner html of the target element
- `outerHTML` - Replace the entire target element with the response
- `beforebegin` - Insert the response before the target element
- `afterbegin` - Insert the response before the first child of the target element
- `beforeend` - Insert the response after the last child of the target element
- `afterend` - Insert the response after the target element
- `delete` - Deletes the target element regardless of the response
- `none` - No swap will be performed

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

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

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

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

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

Listing 1. 29. Replacing the Entire div

```
<div id="main">

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

</div>
```

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

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

Listing 1. 30. 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 opportunities for improvement that we enumerated regarding plain HTML:

- Opportunity 1: We can now issue an HTTP request with *any* element (in this case we are using a button)
- Opportunity 3: 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 opportunity: the requirement that the entire page be replaced.

- Opportunity 4: We can now replace any element we want in our page via transclusion, and we can do so in any manner want

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. 31. A Terrible Idea, But It Demonstrates The Concept!

```
<div id="main">

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

</div>
```

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

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

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

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

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

Listing 1. 32. A Start

```
<div id="main">

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

</div>
```

❶ A trigger with two events

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

There are two problems with this:

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

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

Listing 1. 33. Better!

```
<div id="main">

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

</div>
```

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

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

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

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

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

We can use it like this:

Listing 1. 34. Better!

```
<div id="main">

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

</div>
```

❶ Listen to the event on the `body` tag

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

A nice little keyboard shortcut! Perfect!

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

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

And hey, check it out! With this final attribute, `hx-trigger`, we have addressed the final opportunity for improvement of HTMl that we enumerated at the start of this chapter:

- Opportunity 2: We can use *any* event to trigger an HTTP request

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 whole new world of user interaction possibilities within HTML!

Here is a table summarizing the opportunities to improve HTML and which htmx attributes address them:

Table 1. 1. Opportunities For Improving HTML

Opportunity To Improve HTML	htmx attributes
Any element should be able to make HTTP requests	`hx-get`, `hx-post`, `hx-put`, `hx-patch`, `hx-delete`
Any event should be able to trigger an HTTP request	`hx-trigger`
Any HTTP Action should be available	`hx-put`, `hx-patch`, `hx-delete`
Any place on the page should be replaceable (transclusion)	`hx-target`, `hx-swap`

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. 35. A Simple htmx-Powered Button

```
<div id="main">

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

</div>
```

❶ With an enclosing form tag, all inputs will be submitted with the button's request

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

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

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

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

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

3.7.2. Including inputs

While enclosing all the inputs you want included in a request is the most common approach for including values from inputs in htmx requests, it isn't always possible or desirable: form tags can have layout consequences and simply cannot be placed in some spots in HTML documents. A good example of the latter situation is in table row (`tr`) elements: the `form` tag is not a valid child or parent of table rows, so you can't place a form within or around

an entire row of data in a table.

To address this issue, htmx provides another mechanism for including value in requests: the `hx-include` attribute. The `hx-include` attribute 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. 36. A Simple htmx-Powered Button

```
<div id="main">

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

</div>
```

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

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

3.7.3. Inline Values

A final way to include values in htmx-driven requests is to use the `hx-vals` attribute, which allows you to include "static" values in the request. This can be useful if you have additional information that you want to include in requests, but you don’t want to have this information embedded in, for example, hidden inputs (which would be the standard mechanism for including additional, hidden information in HTML.)

Here is an example of `hx-vals`:

Listing 1. 37. A Simple htmx-Powered Button

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

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

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

For example, if the `state` variable were maintained dynamically, via some JavaScript, and there existed a JavaScript function, `getCurrentState()`, that returned the currently selected state, it could be included dynamically in htmx requests like so:

Listing 1. 38. A Dynamic Value

```
<button hx-get="/contacts" hx-vals='js:{"state":getCurrentState()}'> ❶
  Get The Contacts In The Selected State
</button>
```

❶ With the `js:` prefix, this expression will evaluate at submit time

These three mechanisms, using `form` tags, using the `hx-include` attribute and using the `hx-vals` attribute, allow you to include values in your hypermedia requests with htmx in a manner that should feel very familiar and in keeping with the spirit of HTML, while also giving you the flexibility to achieve what you want.

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. 39. 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. 40. Our trusty button, now with history!

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

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

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

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

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

Drawbacks To The htmx Approach

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

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

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

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

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

As an aside, it's worth nothing that htmx *can* be used to effectively implement a slightly different UX pattern, inline editing, which is often a good alternative to modals, and, in our opinion, is more consistent with the stateless nature of the web. We will look at this approach later in the book, and you can see an example of it here: https://htmx.org/examples/click-to-edit/

3.9. Summary

- Although it has been wildly successful, HTML presents a few opportunities for improvement as a hypermedia

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

4. Putting Hypermedia Into Action

This chapter covers

- Installing htmx in Contact.app
- Adding AJAX-based navigation to Contact.app via "boosting"
- Implementing a proper delete mechanic for contacts
- Validating emails as the user types

- Implementing paging in Contact.app
- Creating a "Click To Load" button
- Adding the "Infinite Scroll" pattern

4.1. Installing htmx

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

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

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

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

Listing 1. 41. Installing htmx

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

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

4.2. Adding AJAX Navigation

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

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

4.2.1. Boosted Links

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

Listing 1. 42. A Boosted Link

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

❶ A simple attribute makes this link AJAX-powered

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

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

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

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

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

4.2.2. Boosted Forms

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

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

Listing 1. 43. A Boosted Form

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

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

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

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

4.2.3. Attribute Inheritance

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

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

Listing 1. 44. A Set of Boosted Links

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

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

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

Listing 1. 45. Boosting Links Via The Parent

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

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

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

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

Listing 1. 46. Disabling Boosting

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

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

❷ The boosting behavior is overridden for this link

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

4.2.4. Progressive Enhancement

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

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

This is not to say that htmx *always* offers progressive enhancement. It is certainly possible to build features that do not offer a "No JS" fallback in htmx, and, in fact, many of the features we will build later in the book will fall into this category. (I will note when a feature is progressive enhancement friendly and when it is not.) Ultimately, it is up to you, the developer, to decide if the tradeoffs of progressive enhancement (more basic UX

functionality, a limited improvement over plain HTML) are worth the benefits for your applications users.

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

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

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

1. Boosting The Entire Contact.app

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

❶ All links and forms will be boosted now!

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

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

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

4.3. Deleting Contacts

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

Listing 1. 47. Plain HTML Form To Delete A Contact

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

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

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

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

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

Listing 1. 48. An htmx Powered Button For Deleting A Contact

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

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

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

4.3.1. Updating The Server Side

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

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

Here is the original code:

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

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

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

HTTP Response Codes

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

HTTP breaks response codes up into various categories:

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

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

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

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

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

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

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

Putting It All Together

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

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

Here is our updated code:

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

❶ A slightly different path and method for the handler

❷ The response code is now a 303

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

4.3.2. Targeting The Right Element

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

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

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

Listing 1. 49. A fixed htmx Powered Button For Deleting A Contact

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

❶ We have added an explicit target to the button now

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

4.3.3. Updating The Location Bar URL Properly

Well, almost.

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

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

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

Listing 1. 50. Deleting A Contact, Now With Proper Location Information

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

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

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

4.3.4. One Last Thing

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

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

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

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

Listing 1. 51. Confirming Deletion

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

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

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

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

4.3.5. Progressive Enhancement?

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

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

4.4. Next Steps: Validating Emails

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

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

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

Listing 1. 52. Server Side Validation On Contact Update

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

❶ We attempt to save the contact

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

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

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

Listing 1. 53. Validation Error Messages

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

❶ Display any errors associated with the email field

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

Server Side Validation

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

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

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

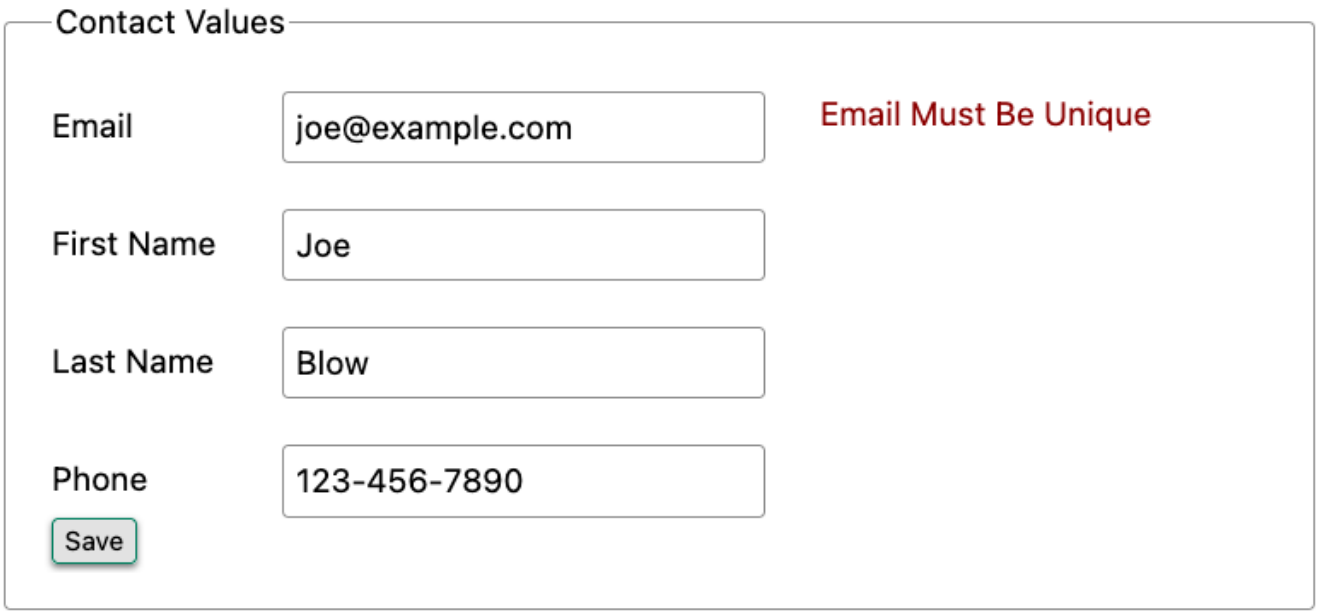


Figure 1. 5. Email Validation Error

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

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

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

4.4.1. Updating Our Input Type

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

Listing 1. 54. Changing The Input To Type `email`

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

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

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

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

Not bad!

Server Side vs. Client Side Validations

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

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

4.4.2. Inline Validation

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

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

Recall the current HTML for our email input:

Listing 1. 55. The Initial Email Configuration

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

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

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

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

Let's make those changes to our HTML:

Listing 1. 56. Our Updated HTML

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

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

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

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

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

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

Here is the code:

Listing 1. 57. Our Email Validation End-Point

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

❶ Look up the contact by id

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

❸ Validate the contact

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

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

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

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

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

4.4.3. Taking Our User Experience Further

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

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

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

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

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

Listing 1. 58. Triggering With `keyup` Events

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

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

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

4.4.4. Debouncing Our Validation Requests

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

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

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

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

Listing 1. 59. Debouncing the `keyup` Event

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

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

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

4.4.5. Ignoring Non-Mutating Keys

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

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

Listing 1. 60. Only Sending Requests When The Input Value Changes

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

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

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

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

4.5. Another Improvement: Paging

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

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

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

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

now for our simple application.

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

Listing 1. 61. Adding Paging Widgets To Our List of Contacts

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

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

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

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

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

Now lets address the server side implementation.

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

Listing 1. 62. Adding Paging To Our Request Handler

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

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

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

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

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

4.5.1. Click To Load

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

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

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

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

Listing 1. 63. Changing To "Click To Load"

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

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

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

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

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

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

Relative Positional Targets

Here we saw the first example of a target that was "relatively positioned": `closest tr`.

The `closest` keyword indicates that the closest parent that matches the following CSS selector is the target. So in this example the target was the `tr` that was enclosing the button.

htmx also supports `next` and `previous` relative positional expressions, allowing you to target the next element or previous element that matches a given CSS selector.

Relative positional expressions like this are quite powerful and allow you to avoid having to generate `id` attributes in your HTML just so you can target a particular element.

4.5.2. Infinite Scroll

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

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

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

Listing 1. 64. Changing To "Infinite Scroll"

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

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

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

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

4.6. Summary

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

5. Hypermedia In Action

6. Advanced Hypermedia Patterns

This chapter covers

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

- Implementing a bulk delete of contacts

6.1. Active Search

So far so good with Contact.app: we have a nice little web application with some significant improvements over a plain HTML-based application. You might be thinking that we've gotten about as much interactivity as the hypermedia approach will allow for without using JavaScript. Is it time wave the white flag and bring out the scripting tools to make any further progress with Contact.app?

Nope, not yet! We still have quite a few interesting and useful interactive UX patterns to add to our application, all of which can be done using good old hypermedia! We *will* eventually add some client-side scripting to our application: hypermedia may be powerful but it isn't *all powerful* and sometimes scripting is the best way to achieve things. But, for now, let's see what we can accomplish without it.

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

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

Let's recall what the current search field in our application currently looks like:

Listing 5. 65. Our Search Form

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

❶ The `q` or "query" paramter our client side code uses to search

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

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

To add active search behavior, we will need to add a few htmx attributes to the search input. We will leave the current form as it is, with an `action` and `method`, so that, in case a user does not have JavaScript enabled, the normal search behavior continues to work. This will make our improvement a nice "progressive enhancement".

Now, in addition to the regular form behavior, we *also* want to issue an HTTP `GET` request when a key up occurs. We want to issue this request to the same URL as the normal form submission. Finally, we only want to do this after a small pause in typing has occurred. Where have we seen that pattern before?

Well, it turns out that this is very similar to the functionality that we needed for email validation isn't it? Indeed it is! We can, in fact, take the `hx-trigger` attribute directly from our email validation example, with its small, 200 millisecond delay to allow a user to stop typing before a request is triggered.

This is a good example of how common patterns come up again and again in htmx.

Listing 5. 66. Adding Active Search Behavior

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

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

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

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

example

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

What we have is pretty close to what we want, but recall that the default target for an element is itself. As things currently stand, an HTTP `GET` request will be issued to the `/contacts` path, which will, as of now, return an entire HTML document of search results! This whole document will then be inserted into the *inner* HTML of an input!

Well, that's pretty meaningless: `input` elements aren't allowed to have any HTML inside of them. The browser will, sensibly, just ignore the htmx request to do this. So, at this point, when a user types anything into our input, a request will be issued (you can see it in your browser development console) but, unfortunately, it will appear to the user as if nothing has happened at all.

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

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

Listing 5. 67. Adding Active Search Behavior

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

❶ Target the `tbody` tag on the page

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

Now if you try typing something into the search box, we'll see some results: a request is made and the results are inserted into the document within the `tbody`. Unfortunately, the content that is coming back is still the entire document. So we end up with a "double render" situation, where an entire document has been inserted *inside* another element, with all the navigation, headers and footers and so forth re-rendered within that element. Not good! But, thankfully, pretty easy to fix.

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

Listing 5. 68. Using `hx-select` for Active Search

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

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

6.1.1. Server Side Tricks With htmx

Let's look at another, more advanced technique for dealing with this situation. Currently, we are letting the server create a full HTML document in response and then, on the client side, filtering it down. This is easy and might be necessary if we don't control the server side or can't easily modify responses.

In this situation, since we are doing "Full Stack" development, where we control both the front end *and* the back end, we have another option: we can modify our server responses. Let's take this opportunity to explore returning different HTML content based on the context information that htmx provides with requests.

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

Listing 5. 69. Server Side Search

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

❶ This is where the search logic happens

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

So how do we want to change this? Well, we want to render two different bits of HTML content *conditionally*:

- If this is a "normal" request for the entire page, we want to render the `index.html` template in the current manner. In fact, we don't want anything to change if this is a "normal" request.
- However, if this is an "Active Search" request, we only want to render just the content that is within the `tbody`, that is, just the table rows of the page.

So we need some way to determine exactly which of these two different types of requests to the `/contact` URL is being made, in order to know exactly which content we want to render.

It turns out that htmx helps us distinguish between these two cases by including a number of HTTP *Request Headers* when it makes requests. Request Headers are a feature of HTTP, allowing clients (e.g. web browsers) to include name/value pairs of metadata associated with requests to help the server understand what the client is requesting.

Here are the HTTP request headers that htmx includes in every request that it makes:

Header	Description
HX-Boosted	This will be the string "true" if the request is made via an element using hx-boost

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

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

Let’s add some conditional logic to our controller to look for that header and, if the value is `search`, we render only the rows rather than the whole `index.html` template:

Listing 5. 70. Updating Our Server Side Search

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

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

❷ We need to learn how to render just the table rows

Next, let's look at how we can render only those rows.

6.1.2. Factoring Your Templates

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

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

Listing 5. 71. The Contacts Table

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

Now, it is the `for` loop in this template that produces all the rows in the final content generated by `index.html`. So, what we want to do is to move the `for` loop and, therefore, the rows it creates out to a *separate template* so that only that little bit of HTML can be rendered independently from `index.html`.

Let's call this new template `rows.html`:

Listing 5. 72. Our New `rows.html` file

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

Using this template we can render only the `tr` elements for a given collection of contacts.

Now, of course, we still want to include this content in the `index.html` template: we are *sometimes* going to be rendering the entire page, and sometimes only rendering the rows. In order to keep `index.html` rendering property, we can include the `rows.html` template by using the Jinja2 `include` directive at the position we want the content from `rows.html` inserted:

Listing 5. 73. Including The New File

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

❶ This directive "includes" the `rows.html` file, inserting its content into the current template

So far, so good: the `/contacts` page still rendering properly, just as it did before we split the rows out of the `index.html` template.

OK, so the last step is to fix up our controller to take advantage of our new `rows.html` template when we are doing an Active Search. Luckily, the update is simple: since `rows.html` is just another template, we only need to call the `render_template` function with it:

Listing 5. 74. Updating Our Server Side Search

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

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

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

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

6.1.3. Updating The Navigation Bar

You may have noticed one shortcoming of our Active Search when compared with submitting the form: when you submit the form it will update the navigation bar of the browser to include the search term. So, for example, if you search for "joe" in the search box, you will end up with a url that looks like this in your browser's nav bar:

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

This is a nice feature of browsers: it allows you to bookmark the search or to copy the URL and send it to someone else. All they have to do is to click on the link, and they will repeat the exact same search. This is also tied in with the browser's notion of history: if you click the back button it will take you to the previous URL that you came from. If you submit two searches and want to go back to the first one, you can simply hit back and the browser will "return" to that search. (It may use a cached version of the search rather than issuing another request, but that's a longer story.)

As it stands right now, during Active Search, we are not updating the browser's navigation bar, so you aren't getting nice copy-and-pasteable links and you arne't getting history entries, so no back button support. Fortunately, htmx provides a way for doing this, the `hx-push-url` attribute.

The `hx-push-url` attribute lets you tell htmx "Please push the URL of this request into the browser's navigation bar". Push might seem like an odd verb to use here, but that's the term that the underlying browser history API uses, which stems from the fact that it models browser history as a "stack" of locations: when you go to a new location, that location is "pushed" onto the stack of history elements, and when you click "back", that location is "popped" off the history stack.

So, to get proper history support for our Active Search, all we need to do is to set the `hx-push-url` attribute to `true`. Let's update our search input:

Listing 5. 75. Updating The URL During Active Search

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

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

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

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

6.1.4. Adding A Request Indicator

A final touch for our Active Search pattern is to add a request indicator to let the user know

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

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

Let's add a spinner after our search input:

Listing 5. 76. Updating The URL During Active Search

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

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

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

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

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

than `opacity`). htmx is flexible in this regard.

Use Request Indicators!

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

Be sure not to neglect this significant aspect of your application! Even though requests might seem instant when you are working on your application locally, in the real world they can take quite a bit longer due to network latency. It's often a good idea to take advantage of browser developer tools that allow you to throttle your local browsers response times. This will give you a better idea of what real world users are seeing, and show you where indicators might help users understand exactly what is going on.

So there we go: we now have a pretty darned sophisticated user experience built out when compared with plain HTML, but we've built it all as a hypermedia-driven feature, no JSON or JavaScript to be seen! This particular implementation also has the benefit of being a progressive enhancement, so this aspect of our application will continue to work for clients that don't have JavaScript enabled. Pretty slick!

6.2. Lazy Loading

With Active Search behind us, let's move on to a very different sort of problem, that of lazy loading. Lazy loading is when the loading of something is deferred until later, when needed. This is commonly used as a performance enhancement: you avoid the processing resources necessary to produce some data until that data is actually needed.

Let's add a count of the total number of contacts below the bottom of our contacts table. This will give us a potentially expensive operation that we can use to demonstrate how easy it is to add lazy loading to our application using htmx.

First let's update our server code in the `/contacts` request handler to get a count of the total number of contacts. We will pass that count through to the template to render some new HTML.

Listing 5. 77. Adding A Count To The UI

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

❶ Get the total count of contacts from the Contact model

❷ Pass the count out to the `index.html` template to use when rendering

As with the rest of the application, in the interest of staying focused on the *hypermedia* part of Contact.app, we are not going to look into the details of how `Contact.count()` works. We just need to know that:

- It returns the total count of contacts in the contact database
- It may potentially be slow

Next lets add some HTML to our `index.html` that takes advantage of this new bit of data, showing a message next to the "Add Contact" link with the total count of users. Here is what our HTML looks like:

Listing 5. 78. Adding A Contact Count Element To The Application

```
<p>
    <a href="/contacts/new">Add Contact</a> <span>({{ count }} total
Contacts)</span>❶
</p>
```

❶ A simple span with some text showing the total number of contacts.

Well that was easy, wasn’t it? Now our users will see the total number of contacts next to

the link to add new contacts, to give them a sense of how large the contact database is. This sort of rapid development is one of the joys of developing web applications the old way.

Here is what the feature looks like in our application:

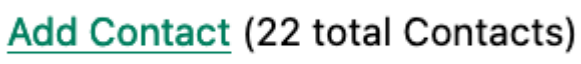


Figure 5. 6. Total Contact Count Display

Beautiful.

Of course, as you probably suspected, all it not perfect. Unfortunately, upon shipping this feature to production, we start getting some complaints from the users that the application "feels slow". So, like all good developers faced with a performance issues, rather than guessing what the issue might be, we try to get a performance profile of the application to see what exactly is causing the problem.

It turns out, surprisingly, that the problem is that innocent looking `Contacts.count()` call, which is taking up to a second and a half to complete. Unfortunately, for reasons beyond the scope of this book, it is not possible to improve that load time, nor it is also not possible to cache the result. This leaves us with two choices:

- Remove the feature
- Come up with some other way to mitigate the performance issue

After talking with your project manager about the various options you have, it becomes clear that removing the feature isn't an acceptable solution: a big customer demanded it as part of a huge enterprise deal and there is no going back now. We will need to take another approach to mitigating this performance issue. The approach we decide to use is Lazy Loading, where we defer loading the contact count "until later".

Let's look at exactly how we can accomplish this using htmx.

6.2.1. Pulling The Expensive Code Out

The first step in implmenting the Lazy Load pattern is to pull the expensive code, that is, the call to `Contacts.count()`. out of request handler for the `/contacts` end point.

Let's pull this call into a handler for a new end point that we will put at `/contacts/count` path instead. Now, for this new end point, we won't need to render a template at all. Its sole job is going to be to render that small bit of text that is in the span, "(22 total Contacts)"

Here is what the new code will look like:

Listing 5. 79. Pulling The Expensive Code Out

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

@app.route("/contacts/count")
def contacts_count():
    count = Contact.count() ❸
    return "(" + str(count) + " total Contacts)" ❹
```

❶ We no longer call `Contacts.count()` in this handler

❷ `count` is no longer passed out to the template to render in the `/contacts` handler

❸ We create a new handler at the `/contacts/count` path that does the expensive calculation

❹ Return the string with the total number of contacts in it

Great! So now we have moved the performance issue out of the `/contacts` handler code and created a new HTTP end point that will produce the expensive-to-create count for us.

The next step is to hook up the span that displays the count to this new path. As we said earlier, the default behavior of htmx is to place any content it receives for a given request into the `innerHTML` of an element, which is exactly what we want here: we want to retrieve this text and put it into the `span`. So we can simply use an `hx-get` attribute

pointing to this new path to do exactly that.

However, recall that the default *event* that will trigger a request for a `span` element in htmx is the `click` event. Well, that's not what we want! Instead, we want this request to trigger immediately, when the page loads. To do this, we can add the `hx-trigger` attribute to update the trigger of the requests for the element, and use the `load` event.

The `load` event is a special event that htmx triggers on all content when it is loaded into the DOM. By setting `hx-trigger` to `load`, we will cause htmx to issue the `GET` request when the `span` element is loaded into the page.

Here is our updated template code:

Listing 5. 80. Adding A Contact Count Element To The Application

```
<p>
    <a href="/contacts/new">Add Contact</a> <span hx-get="/contacts/count" hx-
trigger="load"></span>❶
</p>
```

❶ Issue a `GET` to `/contacts/count` when the `load` event occurs

Note that the `span` starts empty: we have removed the content from it, and we are allowing the request to `/contacts/count` to populate it instead.

And, check it out, our `/contacts` page is fast again! When you navigate to the page it feels very snappy and profiling shows that yes, indeed, the page is loading much more quickly. Why is that? Well, we've deferred the expensive calculation to a secondary request, allowing the initial request to finish loading much more quickly.

You might say "OK, great, but it's still taking a second or two to get the total count on the page." That's true, but often the user may not be particularly interested in the total count. They may just want to come to the page and search for an existing user, or perhaps they may want to edit or add a user. The total count is often just a "nice to have" bit of information in these cases. By deferring the calculation of the count in this manner we let users get on with their use of the application while we perform the expensive calculation.

Yes, the total time to get all the information on the screen takes just as long. (It actually might be a bit longer since we now have two requests that need to get all the information.)

But the *perceived performance* for the end user will be much better: they can do what they want nearly immediately, even if some information isn't available instantaneously. Lazy Loading is a great tool to have in your tool belt when optimizing your web application performance!

6.2.2. Adding An Indicator

Unfortunately there is one somewhat disconcerting aspect to our current implementation: the count is lazily loaded, but there is no way for a user to know that this computation is being done. As it stands, the count just sort of bursts onto the scene whenever the request to `/contacts/count` completes.

That's not ideal. What we want is an indicator, like we added to our active search example. And, in fact, we can simply reuse the same spinner image here!

Now, in this case, we have a one-time request and, once the request is over, we are not going to need the spinner anymore. So it doesn't make sense to use the exact same approach we did with the active search example. Recall that in that case we placed a spinner *after* the span and using the `hx-indicator` attribute to point to it.

In this case, since the spinner is only used once, we can put it *inside* the content of the span. When the request completes the content in the response will be placed inside the span, replacing the spinner with the computed contact count. It turns out that htmx allows you to place indicators with the `htmx-indicator` class on them inside of elements that issue htmx-powered requests. In the absence of an `hx-indicator` attribute, these internal indicators will be shown when a request is in flight.

So let's add that spinner from the active search example as the initial content in our span:

Listing 5. 81. Adding An Indicator To Our Lazily Loaded Content

```
<span hx-get="/contacts/count" hx-trigger="load">
  <img id="spinner" class="htmx-indicator" src="/static/img/spinning-circles.svg"/>❶
</span>
```

❶ Yep, that's it

Great! Now when the user loads the page, rather than having the total contact count sprung on them like a surprise, there is a nice spinner indicating that something is coming. Much

better!

Note that all we had to do was copy and paste our indicator from the active search example into the `span`! This is a great demonstration of how htmx provides flexible, composable features and building blocks to work with: implementing a new feature is often just a copy-and-paste, with maybe a tweak or two, and you are done.

6.2.3. But That's Not Lazy!

You might say "OK, but that's not really lazy. We are still loading the count immediately when the page is loaded, we are just doing it in a second request. You aren't really waiting until the value is actually needed."

Fine. Let's make it *lazy* lazy: we'll only issue the request when the `span` scrolls into view.

To do that, lets recall how we set up the infinite scroll example: we used the `revealed` event for our trigger. That's all we want here, right? When the element is revealed we issue the request?

Yep, that's it! Once again, we can mix and match concepts across various UX patterns to come up with solutions to new problems in htmx.

Listing 5. 82. Making It Lazy Lazy

```
<span hx-get="/contacts/count" hx-trigger="revealed"> ❶
  <img id="spinner" class="htmx-indicator" src="/static/img/spinning-circles.svg"/>
</span>
```

❶ Change the `hx-trigger` to `revealed`

Now we have a truly lazy implementation, deferring the expensive computation until we are absolutely sure we need it. A pretty cool trick, and, again, a simple one-attribute change demonstrates the flexibility of both htmx the hypermedia approach.

6.3. Inline Delete

We now have some pretty slick UX patterns in our application, but let's not rest on our laurels. For our next hypermedia trick, we are going to implement "inline delete", where a contact can be deleted directly from the list view of all contacts, rather than requiring the user to drill in to the edit view of particular contact to access the "Delete Contact" button.

We already have "Edit" and "View" links for each row, in the `rows.html` template:

Listing 5. 83. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
</td>
```

We want to add a "Delete" link as well. And we want that link to act an awful lot like the "Delete Contact" from `edit.html`, don't we? We'd like to issue an HTTP `DELETE` to the URL for the given contact, we want a confirmation dialog to ensure the user doesn't accidentally delete a contact. Here is the "Delete Contact" html:

Listing 5. 84. The Existing Row Actions

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

Is this going to be another copy-and-paste job with a bit of tweaking?

It sure is!

One thing to note is that, in the case of the "Delete Contact" button, we want to rerender the whole screen and update the URL, since we are going to be returning from the edit view for the contact to the list view of all contacts. In the case of this link, however, we are already on the list of contacts, so there is no need to update the URL, and we can omit the `hx-push-url` attribute.

Here is our updated code:

Listing 5. 85. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
    <a href="#" hx-delete="/contacts/{{ contact.id }}"
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="body">Delete</a> ❶
</td>
```

❶ Almost a straight copy of the "Delete Contact" button

As you can see, we have added a new anchor tag and given it a blank target (the `#` value in its `href` attribute) to retain the correct mouse-over styling behavior of the link. We've also copied the `hx-delete`, `hx-confirm` and `hx-target` attributes from the "Delete Contact" button, but omitted the `hx-push-url` attributes since we don't want to update the URL of the browser.

And… that's it! We now have inline delete working, even with a confirmation dialog!

1. A Style Sidebar

One thing is really starting to bother me about our application: we now have quite a few actions stacking up in our contacts table, and it is starting to look very distracting:

Joe	Blow	123-456-7890	joe13@example.com	Edit View Delete
Joe	Blow	123-456-7890	joe14@example.com	Edit View Delete
Joe	Blow	123-456-7890	joe15@example.com	Edit View Delete
Joe	Blow	123-456-7890	joe16@example.com	Edit View Delete
Joe	Blow	123-456-7890	joe17@example.com	Edit View Delete

Figure 5. 7. That's a Lot of Actions

It would be nice if we didn't show the actions all in a row, and it would be nice if we only showed the actions when the user indicated interest in a given row. We will return to this problem after we look at the relationship between scripting and a Hypermedia Driven Application in a later chapter.

For now, let's just tolerate this less-than-ideal user interface, knowing that we will return to it later.

6.3.1. Getting Fancy

We can get even fancier here, however. What if, rather than re-rendering the whole page, we just removed the row for the contact? The user is looking at the row anyway, so is there really a need to re-render the whole page?

To do this, we'll need to do a couple of things:

- We'll need to update this link to target the row that it is in
- We'll need to change the swap to `outerHTML`, since we want to replace (really, remove) the entire row
- We'll need to update the server side to render empty content when the `DELETE` is issued from a row rather than from the "Delete Contact" button on the contact edit page

First things first, update the target of our "Delete" link to be the row that the link is in, rather than the entire body. We can once again take advantage of the relative positional `closest` feature to target the closest `tr`, like we did in our "Click To Load" and "Infinite Scroll" features:

Listing 5. 86. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
    <a href="#" hx-delete="/contacts/{{ contact.id }}"
        hx-swap="outerHTML"
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="closest tr">Delete</a> ❶
</td>
```

❶ Updated to target the closest enclosing `tr` (table row) of the link

6.3.2. Updating The Server Side

Now we need to update the server side as well. We want to keep the "Delete Contact" button working as well, and in that case the current logic is correct. So we'll need some way to differentiate between `DELETE` requests that are triggered by the button and `DELETE` requests that come from this anchor.

The cleanest way to do this is to add an `id` attribute to the "Delete Contact" button, so that

we can inspect the `HX-Trigger` HTTP Request header to determine if the delete button was the cause of the request. This is a simple change to the existing HTML:

Listing 5. 87. Adding an `id` to the "Delete Contact" button

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

❶ An `id` attribute has been added to the button

With this in place, we now have a mechanism for differentiating between the delete button in the `edit.html` template and the delete links in the `rows.html` template. We can write code very similar to what we did for the active search pattern, using a conditional on the `HX-Trigger` header to determine what we want to do. If that header has the value `delete-btn`, then we know the request came from the button on the edit page, and we can do what we are currently doing: delete the contact and redirect to `/contacts` page.

If it does not have that value, then we can simple delete the contact and return an empty string. This empty string will replace the target, in this case the row for the given contact, thereby removing the row from the UI.

Let's make that change to our server side code:

Listing 5. 88. Updating Our Server Code To Handle Two Different Delete Patterns

```
@app.route("/contacts/<contact_id>", methods=["DELETE"])
def contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete()
    if request.headers.get('HX-Trigger') == 'delete-btn': ❶
        flash("Deleted Contact!")
        return redirect("/contacts", 303)
    else:
        return "" ❷
```

❶ If the delete button on the edit page submitted this request, then continue to do the logic

we had previous

❷ If not, simply return an empty string, which will delete the row

Believe it or not, we are now done: when a user clicks "Delete" on a contact row and confirms the delete, the row will disappear from the UI. Poof! Once again, we have a situation where just changing a few lines of simple code gives us a dramatically different behavior. Hypermedia is very powerful!

6.3.3. Getting Super Fancy With The htmx Swapping Model

This is pretty cool, but there is another improvement we can make if we take some time to understand the htmx content swapping model: it sure would be exciting if, rather than just instantly deleting the row, we faded it out before we removed it. That easement makes it more obvious that the row is being removed, giving the user some nice visual feedback on the deletion.

It turns out we can do this pretty easily with htmx, but to do so we'll need to dig in to exactly how htmx swaps content.

The htmx Swapping Model

You might think that htmx simply puts the new content into the DOM, but that's not in fact how it works. Instead, content goes through a series of steps as it is added to the DOM:

- When content is received and about to be swapped into the DOM, the `htmx-swapping` CSS class is added to the target element
- A small delay then occurs (we will discuss why this delay exists in a moment)
- Next, the `htmx-swapping` class is removed from the target and the `htmx-settling` class is added
- The new content is swapped into the DOM
- Another small delay occurs
- Finally, the `htmx-settling` class is removed from the target

There is more to the swap mechanic (settling, for example, is a more advanced topic that we will discuss in a later chapter) but for now this is all you need to know about it.

Now, there are small delays in the process here, typically on the order of a few milliseconds. Why so? It turns out that these small delays allow *CSS transitions* to occur.

CSS transitions are a technology that allow you to animate a transition from one style to another. So, for example, if you changed the height of something from 10 pixels to 20 pixels, by using a CSS transition you can make the element smoothly animate to the new height. These sorts of animations are fun, often increase application usability, and are a great mechanism to add polish and fit-and-finish to your web application.

Unfortunately, CSS transitions are not available in plain HTML: you have to use JavaScript and add or remove classes to get them to trigger. This is why the htmx swap model is more complicated than you might initially think: by swapping in classes and adding small delays, you can access CSS transitions purely within HTML, without needing to write any JavaScript!

6.3.4. Taking Advantage of `htmx-swapping`

OK, so, let's go back and look at our inline delete mechanic: we click an htmx enhanced link which deletes the contact and then swaps some empty content in for the row. We know that, before the `tr` element is removed, it will have the `htmx-swapping` class added to it. We can take advantage of that to write a CSS transition that fades the opacity of the row to 0. Here is what that CSS looks like:

Listing 5. 89. Adding A Fade Out Transition

```
tr.htmx-swapping { ❶
  opacity: 0; ❷
  transition: opacity 1s ease-out; ❸
}
```

❶ We want this style to apply to `tr` elements with the `htmx-swapping` class on them

❷ The `opacity` will be 0, making it invisible

❸ The `opacity` will transition to 0 over a 1 second time period, using the `ease-out` function

Again, this is not a CSS book and I am not going to go deeply into the details of CSS transitions, but hopefully the above makes sense to you, even if this is the first time you've seen CSS transitions.

So, think about what this means from the htmx swapping model: when htmx gets content back to swap into the row it will put the `htmx-swapping` class on the row and wait a bit. This will allow the transition to a zero opacity to occur, fading the row out. Then the new (empty) content will be swapped in, which will effectively removing the row.

Sounds good, and we are nearly there. There is one more thing we need to do: the default "swap delay" for htmx is very short, a few milliseconds. That makes sense in most cases: you don't want to have much of a delay before you put the new content into the DOM. But, in this case, we want to give the CSS animation time to complete before we do the swap, we want to give it a second, in fact.

Fortunately htmx has an option for the `hx-swap` annotation that allows you to set the swap delay: following the swap type you can add `swap:` followed by a timing value to tell htmx to wait a specific amount of time before it swaps. Let's update our HTML to allow a one

second delay before the swap is done for the delete action:

Listing 5. 90. The Existing Row Actions

```
<td>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
    <a href="#" hx-delete="/contacts/{{ contact.id }}"
        hx-swap="outerHTML swap:1s" ❶
        hx-confirm="Are you sure you want to delete this contact?"
        hx-target="closest tr">Delete</a>
</td>
```

❶ A swap delay changes how long htmx waits before it swaps in new content

With this modification, the existing row will stay in the DOM for an additional second, with the `htmx-swapping` class on it. This will give the row time to transition to an opacity of zero, giving the fade out effect we want.

Now, when a user clicks on a "Delete" link and confirms the delete, the row will slowly fade out and then, once it has faded to a 0 opacity, it will be removed. Fancy! And all done in a declarative, hypermedia oriented manner, no JavaScript required! (Well, obviously htmx is written in JavaScript, but you know what I mean: we didn't have to write any JavaScript to implement the feature!)

6.4. Bulk Delete

Our final feature in this chapter is going to be a "Bulk Delete" feature. The current mechanism for deleting users is nice, but it would be annoying if a user wanted to delete five or ten contacts at a time, wouldn't it? For the bulk delete feature, we'll add the ability to select rows via a checkbox input and delete them all in a single go by clicking a "Delete Selected Contacts" button.

To get started with this feature, we'll need to add a checkbox input to each row in the `rows.html` template. This input will have the name `contacts` and its value will be the `id` of the contact for the current row. Here is what the updated code for `rows.html` looks like:

Listing 5. 91. Adding A Checkbox To Each Row

```
{% for contact in contacts %}
<tr>
  <td><input type="checkbox" name="selected_contact_ids" value="{{ contact.id
}}"></td> ❶
  <td>{{ contact.first }}</td>
  ... omitted
</tr>
{% endfor %}
```

❶ A new cell with the checkbox input whose value is set to the current contact's id

We'll also need to add an empty column in the header for the table to accommodate the checkbox column. With that done we now get a series of check boxes, one for each row, a pattern no doubt familiar to you from the web:

☐	Joe	Blow	123-456-7890	joe9@example.com	Edit View Delete
☐	Joe	Blow	123-456-7890	joe10@example.com	Edit View Delete
☐	Joe	Blow	123-456-7890	joe11@example.com	Edit View Delete
☐	Joe	Blow	123-456-7890	joe12@example.com	Edit View Delete

Figure 5. 8. Checkboxes For Our Contact Rows

If you are not familiar with or have forgotten the way checkboxes work in HTML: a checkbox will submit its value associated with the name of the input if and only if it is checked. So if, for example, you checked the contacts with the ids 3, 7 and 9, then those three values would all be submitted to the server. Since all the checkboxes in this case have the same name, `contacts`, all three values would be submitted with the name `contacts`.

6.4.1. The "Delete Selected Contacts" button

The next step is to add a button below the table that will delete all the selected contacts. We want this button, like our delete links in each row, to issue an HTTP `DELETE`, but rather than issuing it to the URL for a given contact, like we do with the inline delete links and with the delete button on the edit page, here we want to issue the delete to the `/contacts` URL. As with the other delete elements, we want to confirm that the user wishes to delete the contacts, and, for this case, we are going to target the body of page, since we are going to rerender the whole table.

Here is what the template code looks like:

Listing 5. 92. The Delete Selected Contacts Button.

```
<button hx-delete="/contacts" ❶
        hx-confirm="Are you sure you want to delete these contacts?" ❷
        hx-target="body"> ❸
    Delete Selected Contacts
</button>
```

❶ Issue a `DELETE` to `/contacts`

❷ Confirm that the user wants to delete the selected contacts

❸ Target the body

Great, pretty easy. One question though: how are we going to include the values of all the selected checkboxes in the request? As it stands right now, this is just a stand-alone button, and it doesn't have any information indicating that it should include any other information. Fortunately, htmx has a few different ways to include values of inputs with a request.

One way would be to use the `hx-include` attribute, which allows you to use a CSS selector to specify the elements you want to include in the request. That would work fine here, but we are going to use another approach that is a bit simpler in this case. By default, if an element is a child of a `form` element, htmx will include all the values of inputs within that form. In situations like this, where there is a bulk operation for a table, it is common to enclose the whole table in a form tag, so that it is easy to add buttons that operate on the selected items.

Let's add that form tag around the form, and be sure to enclose the button in it as well:

Listing 5. 93. The Delete Selected Contacts Button.

```
    <form> ❶
        <table>
          ... omitted
        </table>
        <button hx-delete="/contacts"
                hx-confirm="Are you sure you want to delete these contacts?"
                hx-target="body">
            Delete Selected Contacts
        </button>
    </form> ❷
```

❶ The form tag encloses the entire table

❷ And also encloses the button

Now, when the button issues a `DELETE`, it will include all the contact ids that have been selected as the `selected_contact_ids` request variable. Great!

6.4.2. The Server Side for Delete Selected Contacts

The server side implementation is going to look an awful lot like our original server side code for deleting a contact. In fact, once again, we can just copy and paste, and fix a bit of stuff up:

- We want to change the URL to `/contacts`
- We want the handler to get *all* the ids submitted as `selected_contact_ids` and iterate over each one, deleting the given contact

Those are really the only changes we need to make! Here is what the server side code looks like:

Listing 5. 94. The Delete Selected Contacts Button.

```
@app.route("/contacts/", methods=["DELETE"]) ❶
def contacts_delete_all():
    contact_ids = list(map(int, request.form.getlist("selected_contact_ids"))) ❷
    for contact_id in contact_ids: ❸
        contact = Contact.find(contact_id)
        contact.delete() ❹
    flash("Deleted Contacts!") ❺
    contacts_set = Contact.all()
    return render_template("index.html", contacts=contacts_set)
```

❶ We handle a `DELETE` request to the `/contacts/` path

❷ We convert the `selected_contact_ids` values submitted to the server from a list of strings to a list integers

❸ We iterate over all of the ids

❹ And delete the given contact with each id

❺ Beyond that, it's the same code as our original delete handler: flash a message and render the `index.html` template

So, as you can see, we just took the original delete logic and slightly modified it to deal with an array of ids, rather than a single id. Readers with sharp eyes might notice one other small change: we did away with the redirect that was in the original delete code. We did so because we are already on the page we want to rerender, so there is no reason to redirect and have the URL update to something new. We can just rerender the page, and the new list of contacts (sans the contacts that were deleted) will be re-rendered.

And there we go, we now have a bulk delete feature for our application. Once again, not a huge amount of code, and we are implementing these features entirely by exchanging hypermedia with a server in the traditional, RESTful manner of the web. Cool!

6.5. Summary

- In this chapter dove into some more advanced user interface features using htmx and hypermedia
- We implemented a nifty "Active Search" feature, allowing users to type and immediately filter down the contacts list
- We then introduced and fixed a performance issue by using the "Lazy Loading" pattern, which defers a calculation until after the initial request for better perceived performance
- Next we implemented an "Inline Delete" feature that allows users to delete contacts directly from the list view, complete with a nice fade out effect
- Finally, we implemented a "Bulk Delete" feature that allows users to select multiple contacts and delete them all with a single click

7. Hypermedia In Action

8. Hyperview: A Mobile Hypermedia

In this chapter, we will:

- Discuss shortcomings with the current state of mobile app development
- Explain how a Hypermedia architecture can address those problems
- Introduce Hyperview as a Hypermedia for mobile application development
- Dive into HXML to show the power of its syntax

8.1. State of mobile app development

In web development, you deliver JS,HTML,CSS,JSON on-demand to a standard web browser. This is not the case with a native mobile app.

With native mobile apps, you compile code into an executable binary targeting the underlying operating system (Android or iOS). This binary gets uploaded and approved through app stores controlled by Google and Apple. When you install or update an app, you're downloading the binary and running the code directly on your device's OS. In this way, mobile apps have a lot in common with old-school desktop apps for Mac or Windows.

There is one important difference between PC desktop apps of yesteryear and today's mobile apps. These days, almost all mobile apps are "networked". By networked, I mean the app needs to read and write data over the Internet to implement core functionality. What's the easiest way to make a mobile app networked? Well, the app stores already require developer to write & distribute code that runs on the user's device. So why not have that frontend code make requests to a backend, and update the UI based on the response?

Thus, developers are naturally led into a SPA-like pattern for developing mobile apps:

- The frontend on the mobile device, runs code to create and update the UI
- The backend is an API, called by the frontend

Just like with SPAs on the web, this architecture has a big downside: the app's logic gets spread across the frontend and backend. Sometimes, logic needs to be duplicated (like to validate form data). Other times, the frontend and backend each implement one part of the app's overall logic. To understand what the app does, a developer needs to trace interactions between two very different codebases.

There's another downside that affects mobile apps more than SPAs: API churn. Remember, the app stores control how your app gets distributed and updated. Users can even control if and when they get updated versions of your app. As a mobile developer, you can't assume that every user will be on the latest version of your app. Your frontend code gets fragmented across many versions, and now your backend needs to support all of them.

8.2. Hypermedia for Mobile Apps

We've seen that the hypermedia architecture can address the problems of SPAs on the web.

But can hypermedia work for mobile apps as well? The answer is yes!

Just like on the web, we can use Hypermedia formats on mobile and let it serve as the engine of application state. All of the logic is controlled from the backend, rather than being spread between two codebases. Hypermedia architecture also solves the annoying problem of API churn on mobile apps. Since the backend servs a Hypermedia response containing both data and actions, there's no way for the data and UI to get out of sync. No more worries about backwards compatibility or maintaining multiple API versions.

So how can you use Hypermedia for your mobile app? There are two approaches employing hypermedia to build & ship native mobile apps today:

- Web views, which wraps the trusty web platform in a mobile app shell
- Hyperview, a new hypermedia format I designed specifically for mobile apps

8.2.1. Web views

The simplest way to use hypermedia on mobile is to make a web app! Both Android and iOS SDKs provide "web views": chromeless web browsers tbat can be embedded in native apps. Tools like Apache Cordova make it easy to take the URL of a website, and spit out native iOS and Android apps based on web views. If you already have a responsive web app, this gives you "native" Hypermedia apps for free. Sounds too good to be true, right?

Of course, there is a fundamental limitation with this approach. The web and mobile are different platforms, with different capabilities and UX conventions. HTML doesn't natively support common UI patterns of mobile apps. One of the biggest differences is around how each platform handles navigation. On the web, navigation is page-based, with one page replacing another and the browser providing back/forward buttons to navigate the page history. On mobile, navigation is more complex, and tuned for the physicality of gesture-based interactions.

- To drill down, screens slide on top of each other, forming stacks of screens.
- A nav bar at the bottom of the app allows switching between various stacks of screens.
- Modals slide up from the bottom of the app, covering the other stacks and the nav bar.
- Unlike with web pages, all of these screens are still present in memory, rendered and updating based on app state.

The navigation architecture is a major difference between how mobile and web apps function. But it’s not the only one. Many other UX patterns are present in mobile apps, but are not natively supported on the web:

- pull-to-refresh to refresh content in a screen
- horizontal swipe on UI elements to reveal actions
- sectioned lists with sticky headers

While these interactions are not natively supported by web browsers, they can be simulated with JS libraries. Of course, these libraries will never have the same feel and performance as native gestures. And using these libraries usually requires embracing a JS-heavy SPA architecture. This puts us back at square 1! We wanted to avoid using the typical SPA architecture of native mobile app development, so we turned to a Web view. The web view allows us to use good-old hypermedia-based HTML. But to get the desired look & feel of a mobile app, we end up building a SPA in JS, losing the benefits of Hypermedia in the process.

Many mobile apps on iOS and Android are implemented as web views, and they work perfectly fine. If you already have a responsive web app, perhaps wrapping it in a Web view and distributing it through app stores is an easy proposition. But I tend to believe: "let the web be the web, and let mobile be mobile". Your responsive web app already works perfectly fine when accessed through Mobile Chrome or Mobile Safari. Progressive web app support has come a long way on iOS and Android. Users can "install" your web app to their home screen, and launch it with a single tap. Does your web app need to be in the app store too?

To build a hypermedia-based mobile app that feels and acts native, HTML isn’t going to cut it. We need a format designed to represent the interactions and patterns of native mobile apps. That’s exactly what Hyperview does.

8.2.2. Hyperview

Hyperview is an open-source framework that provides:

- A pre-defined hypermedia format for defining mobile apps called HXML
- A hypermedia client for HXML that can be embedded in an app binary on iOS and Android.

- Extension points in HXML and the client to customize the experience on a per-app basis.

The Format

HXML was designed to feel familiar to web developers, used to working with HTML. Thus the choice of XML for the base format. In addition to familiar ergonomics, XML is compatible with server-side rendering libraries. For example, Jinja2 is perfectly suited as a templating library to render HXML, as we'll see in the next chapter. The familiarity of XML and the ease of integration on the backend make it simple to adopt in both new and existing codebases.

```
<doc xmlns="https://hyperview.org/hyperview">
  <screen>
    <styles />
    <body>
      <header>
        <text>My first app</text>
      </header>
      <view>
        <text>Hello World!</text>
      </view>
    </body>
  </screen>
</doc>
```

htmx fill in the "missing parts of HTML" to enable rich web app experiences. If HTML was designed today, I believe the ideas of htmx would be part of the standrd spec, and natively supported by every browser without the need for a 3rd party JS library. Well, HXML was designed today! The HXML format has built-in support for htmx-like interactions. Specifically, HXML is not limited to "click to navigate" and "press to submit" interactions like basic HTML. It supports a range of triggers and actions for modifying the content on a screen. These interactions are bundled together in a powerful concept of "behaviors". Developers can even define new behavior actions to add new capabilities to their app, without the need for scripting. We will learn more about behaviors in later sections of this chapter.

The client

Web developers are lucky. They can assume their users have access to a web browser

capable of rendering any web app. In Hypermedia terms, the Hypermedia (HTML) client is already built and distributed to users. Half the work is done! The developer has to only build the backend to serve Hypermedia responses.

< diagram showing many browsers, pointing to one backend >

This is possible because the web is an open ecosystem built on standards. Any developer can build and host a web app, and any user can access it directly.

As we know, that's not the case with mobile platforms. There is no open standard for building and distributing native mobile apps. And there's definitely no widely distributed "HXML browser". So how can a developer deliver a Hypermedia mobile app using HXML? Well, unlike on the web, the mobile developer must provide both the backend to serve HXML, and a mobile client app to render those HXML responses.

< diagram showing one Hypermedia client, pointing to one backend >

It would be a lot to ask from developers to write their own HXML client. That's why Hyperview provides an open-source client library, written in React Native. This library can be used to bootstrap a new mobile app, or it can be embedded in an existing app. In either case, developers get a full "HXML browser" without needing to write it from scratch.

At first, it might seem like the Hyperview approach requires extra work to write and maintain the mobile app client. But there is a benefit when the developer controls both the server and client. Did you ever wish you could fix a web browser bug? Or maybe add a new HTML element or features to the browser itself? The open nature of the web means that progress happens slowly. New features go through a lengthly standardization process. Browser vendors may prioritize bugs and features that don't match your individual priorities. As a web developer, you may need to wait years until browsers support the feature you need. Or, you can try to work around it with some kludgy JS.

Well, with Hyperview, there is no standards body or lengthly process for new features. As a Hyperview developer, you control your backend and mobile app client. Do you want to add a new element to HXML? Go right ahead! In fact, the Hyperview client library was built with extensibility in mind. There are extension points for custom UI elements and custom behaviors.

By extending the format and client itself, there's no need for Hyperview to include a

scripting layer in HTMX. Features that require client-side logic get "built-in" to the client browser. HTMX responses remain pure, with UI and interactions represented in declarative XML.

8.2.3. Which Hypermedia architecture should I use?

We've discussed two approaches for creating mobile apps using Hypermedia architecture:

- create a backend that returns HTML, and serve it in a mobile app through a web view
- create a backend that returns HXML, and serve it in a mobile app with the Hyperview client

I purposefully described the two approaches in a way to highlight their similarities. After all, they are both using the Hypermedia architecture, just with different formats and clients. Both approaches solve the fundamental issues with traditional, SPA-like mobile app development:

- The backend controls the full state of the app.
- Our app's logic is all in one place.
- The app always runs the latest version, there's no API churn to worry about

So which approach should you use for a Hypermedia-driven mobile app? Based on my experience building both types of apps, I strongly believe the Hyperview results in a better user experience. The web-view will always feel out-of-place on iOS and Android; there's just no good way to replicate the patterns of navigation and interaction that mobile users expect. Hyperview was created specifically to address the limitations of thick-client and web view approaches. After the initial investment to learn Hyperview, you'll get all of the benefits of the Hypermedia architecture, without the downsides of a degraded user experience.

Of course, if you already have a simple, mobile-friendly web app, then using a web-view approach is sensible. You will certainly save time from not having to serve your app as HXML in addition to HTML. But as I will show at the end of this chapter, it doesn't take a lot of work to convert an existing Hypermedia-driven web app into a Hyperview mobile app. But before we get there, we need to introduce the concepts of elements and behaviors in Hyperview. Then, we'll re-build our contacts app in Hyperview.

8.3. Introduction to HXML

8.3.1. Hello World!

HXML was designed to feel natural to web developers coming from HTML. Let's take a closer look at the "Hello World" app defined in HXML:

```
<doc xmlns="https://hyperview.org/hyperview"> ❶
  <screen> ❷
    <styles />
    <body> ❸
      <header> ❹
        <text>My first app</text>
      </header>
      <view> ❺
        <text>Hello World!</text> ❻
      </view>
    </body>
  </screen>
</doc>
```

❶ The root element of the HXML app

❷ The element representing a screen of the app

❸ The element representing the UI of the screem

❹ The element representing the top header of the screen

❺ A wrapper element around the content shown on the screen

❻ The text content shown on the screen

Nothing too strange here, right? The base syntax should be immediately familiar. Just like HTML, the syntax defines a tree of elements using start tags (`<screen>`) and end tags (`</screen>`). Elements can contain other elements (`<view>`) or text (`Hello World!`). Elements can also be empty, represented with an empty tag (`<styles />`). However, you'll notice that the names of the HXML element are different from those in HTML. Let's take a closer look at each of those elements to understand what they do.

`<doc>` is the root of the HXML app. Think of it as equivalent to the `<html>` element in HTML. Note that the `<doc>` element contains an attribute `xmlns="https://hyperview.org/hyperview"`. This defines the default namespace

for the doc. Namespaces are a feature of XML that allow one doc to contain elements defined by different developers. To prevent conflicts when two developers use the same name for their element, each developer defines a unique namespace. We will talk more about namespaces when we discuss custom elements & behaviors later in this chapter. For now, it's enough to know that elements in an HXML doc without an explicit namespace are considered to be part of the `https://hyperview.org/hyperview` namespace.

`<screen>` represents the UI that gets rendered on a single screen of a mobile app. It's possible for one `<doc>` to contain multiple `<screen>` elements, but we won't get into that now. Typically, a `<screen>` element will contain elements that define the content and styling of the screen.

`<styles>` defines the styles of the UI on the screen. We won't get too much into styling in Hyperview in this chapter. Suffice it to say, unlike HTML, Hyperview does not use a separate language (CSS) to define styles. Instead, styling rules such as colors, spacing, layout, and fonts are defined in HXML. These rules are then explicitly referenced by UI elements, much like using classes in CSS.

`<body>` defines the actual UI of the screen. The body includes all text, images, buttons, forms, etc that will be shown to the user. This is equivalent to the `<body>` element in HTML.

`<header>` defines the header of the screen. Typically in mobile apps, the header includes some navigation (like a back button), and the title of the screen. It's useful to define the header separately from the rest of the body. Some mobile OSes will use a different transition for the header than the rest of the screen content.

`<view>` is the basic building block for layouts and structure within the screen's body. Think of it like a `<div>` in HTML. Note that unlike in HTML, a `<div>` cannot directly contain text.

`<text>` elements are the only way to render text in the UI. In this example, "Hello World" is contained within a `<text>` element.

That's all there is to define a basic "Hello World" app in HXML. Of course, this isn't very exciting. Let's cover some other built-in display elements.

8.3.2. UI Elements

Lists

A very common pattern in mobile apps is to scroll through a list of items. The physical properties of a phone screen (long & vertical) and the intuitive gesture of swiping a thumb up & down makes this a good choice for many screens.

HXML has dedicated elements for representing lists and items.

```
<list> ❶
  <item key="item1"> ❷
    <text>My first item</text> ❸
  </item>`
  <item key="item2">
    <text>My second item</text>
  </item>
</list>
```

❶ Element representing a list

❷ Element representing an item in the list, with a unique key

❸ The content of the item in the list.

Lists are represented with two new elements. The `<list>` wraps all of items in the list. It can be styled like a generic `<view>` (width, height, etc). A `<list>` element only contains `<item>` elements. Of course, these represent each unique item in the list. Note that `<item>` is required to have a `key` attribute, which is unique among all items in the list.

You might be asking, "Why do we need a custom syntax for lists of items? Can't we just use a bunch of `<view>` elements?". Yes, for lists with a small number of items, using nested `<views>` will work quite well. However, often the number of items in a list can be long enough to require optimizations to support smooth scrolling interactions. Consider browsing a feed of posts in a social media app. As you keep scrolling through the feed, it's not unusual for the app to show hundreds if not thousands of posts. At any time, you can flick your finger to scroll to almost any part of the feed. Mobile devices tend to be memory-constrained. Keeping the fully-rendered list of items in memory could consume more resources than available. That's why both iOS and Android provide APIs for optimized list UIs. These APIs know which part of the list is currently on-screen. To save memory, they

clear out the non-visible list items, and recycle the item UI objects to conserve memory. By using explicit `<list>` and `<item>` elements in HXML, the Hyperview client knows to use these optimized list APIs to make your app more performant.

It's also worth mentioning that HXML supports section lists. Section lists are useful for building screens like a list of contacts, where contacts are alphabetized and grouped in a section by their starting letter.

```
<section-list> ❶
  <section> ❷
    <section-title> ❸
      <text>A</text>
    </section-title>
    <item key="1"> ❹
      <text>Aaron</text>
    </item>
    <item key="2">
      <text>Adam</text>
    </item>
  </section>

  <section> ❺
    <section-title>
      <text>B</text>
    </section-title>
    <item key="3">
      <text>Bart</text>
    </item>
  </section>
</section-list>
```

❶ Element representing a list with sections

❷ The first section of contacts with names that begin with the letter "A"

❸ Element for the title of the section, rendering text of the letter "A"

❹ An item representing a contact that belongs to this section.

❺ A section for contacts with names that begin with "B".

You'll notice a couple of differences between `<list>` and `<section-list>`. The section list element only contains `<section>` elements, representing a group of items. A section

can contain a `<section-title>` element. This is used to render some UI that acts as the header of the section. This header is "sticky", meaning it stays on screen while scrolling through items that belong to the corresponding section. Finally, `<item>` elements act the same as in the regular list, but can only appear within a `<section>`.

Images

Showing images in Hyperview is pretty similar to HTML, but there are a few differences.

```
<image source="/profiles/1.jpg" style="avatar" />
```

The `source` attribute specifies how to load the image. Like in HTML, the source can be an absolute or relative URL. Additionally, the source can be an encoded data URI, for example ``. However, the source can also be a "local" URL, refering to an image that is bundled as an asset in the mobile app. The local URL is prefixed with `./`:

```
<image source="./logo.png" style="logo" />
```

Using Local URLs is an optimization. Since the images are on the mobile device, they don’t require a network request and will appear quickly. However, bundling the image with the mobile app binary increases the binary size. Using local images is a good tradeoff for images that are frequently accessed but rarely change. Good examples include the app logo, or common button icons.

The other thing to note is the presence of the `style` attribute on the `<image>` element. In HXML, images are required to have a style that has rules for the image’s `width` and `height`. This is different from HTML, where `<img>` elements do not need to explicitly set a width and height. Web browsers will re-flow the content of a web page once the image is fetched and the dimensions are known. While re-flowing content is a reasonable behavior for web-based documents, users do not expect mobile apps to re-flow as content loads. To maintain a static layout, HXML requires the dimensions to be known before the image loads.

8.3.3. Inputs

There’s a lot to cover about inputs in Hyperview. Since this is meant to be an introduction

and not an exhaustive resource, I'll highlight just a few types of inputs. Let's start with an example of the simplest type of input, a text field.

```
<text-field
  name="first_name" ❶
  style="input" ❷
  value="Adam" ❸
  placeholder="First name" ❹
/>
```

❶ The name used when serializing data from this input

❷ The style class applied to the UI element

❸ The currently value set in the field

❹ A placeholder to display when the value is empty

This element should feel familiar to anyone who's created a text field in HTML. One difference is that most inputs in HTML use the `<input>` element with a `type` attribute, eg `<input type="text">`. In Hyperview, each input has a unique name, in this case `<text-field>`. By using different names, we can use more expressive XML to represent the input.

For example, let's consider a case where we want to render a UI that lets the user select one among several options. In HTML, we would use a radio button input, something like `<input type="radio" name="choice" value="option1" />`. Each choice is represented as a unique input element. This never struck me as ideal. Most of the time, radio buttons are grouped together to affect the same name. The HTML approach leads to a lot of boilerplate (duplication of `type="radio"` and `name="choice"` for each choice). Also, unlike radio buttons on desktop, mobile OSes don't provide a strong standard UI for selecting one option. Most mobile apps use richer, custom UIs for these interactions. So in HXML, we implement this UI using an element called `<select-single>`:

```
<select-single name="choice"> ❶
  <option value="option1"> ❷
    <text>Option 1</text> ❸
  </option>
  <option value="option2">
    <text>Option 2</text>
  </option>
</select-single>
```

❶ Element representing an input where a single choice is selected. The name of the selection is defined once here.

❷ Element representing one of the choices. The choice value is defined here.

❸ The UI of the selection. In this example, we use text, but we can use any UI elements.

The `<select-single>` element is the parent of the input for selecting one choice out of many. This element contains the `name` attribute used when serializing the selected choice. `<option>` elements within `<select-single>` represent the available choices. Note that each `<option>` element has a `value` attribute. When pressed, this will be the selected value of the input. The `<option>` element can contain any other UI elements within it. This means that we're not hampered by rendering the input as a list of radio buttons with labels. We can render the options as radios, tags, images, or anything else that would be intuitive for our interface. HXML styling supports modifiers for pressed and selected states, letting us customize the UI to highlight the selected option.

Like I mentioned, describing all features of inputs in HXML would take an entire chapter. Instead, I'll summarize a few other input elements and their features.

- `<select-multiple>` works like `<select-single>`, but it supports toggling multiple options on & off. This replaces checkbox inputs in HTML.
- The `<switch>` element renders a on/off switch that is common in mobile UIs
- The `<date-field>` element supports entering in specific dates, and comes with a wide range of customizations for formatting, settings ranges, etc.

Two more things to mention about inputs. First is the `<form>` element. The `<form>` element is used to group together inputs for serialization. When a user takes an action that triggers a backend request, the Hyperview client will serialize all inputs in the surrounding

<form> and include them in the request. This is true for both GET and POST requests. We will cover this in more detail when talking about behaviors later in this chatper. Also later in this chapter, I'll talk about support for custom elements in HXML. With custom elements, you can also create your own input elements. Custom input elements allow you to build incredible powerful interactions with simple XML syntax that integrates well with the rest of HXML.

8.3.4. Styling

So far, we haven't mentioned how to apply styling to all of the HXML elements. We've seen from the Hello World app that each <screen> can contain a <styles> element. Let's re-visit the Hello World app and fill out the <styles> element.

```
<doc xmlns="https://hyperview.org/hyperview">
  <screen>
    <styles> ❶
      <style class="body" flex="1" flexDirection="column" /> ❷
      <style class="header" borderBottomWidth="1" borderBottomColor="#ccc" />
      <style class="main" margin="24" />
      <style class="h1" fontSize="32" />
      <style class="info" color="blue" />
    </styles>

    <body style="body"> ❸
      <header style="header">
        <text style="info">My first app</text>
      </header>
      <view style="main">
        <text style="h1 info">Hello World!</text> ❹
      </view>
    </body>
  </screen>
</doc>
```

❶ Element encapsulating all of the styling for the screen

❷ Example of a definition of a style class for "body"

❸ Applying the "body" style class to a UI element

❹ Example of applying multiple style classes (h1 and info) to an element

You'll note that in HXML, styling is part of the XML format, rather than using a separate

language like CSS. However, we can draw some parallels between CSS rules and the `<style>` element. A CSS rule consists of a selector and declarations. In the current version of HXML, the only available selector is a class name, indicated by the `class` attribute. The rest of the attributes on the `<style>` element are declarations, consisting of properties and property values.

UI elements within the `<screen>` can reference the `<style>` rules by adding the class names to their `<style>` property. Note the `<text>` element around "Hello World!" references two style classes: `h1` and `info`. The styles from the corresponding classes are merged together in the order they appear on the element. It's worth noting that styling properties are similar to those in CSS (color, margins/padding, borders, etc). Currently, the only available layout engine is based on flexbox.

Styles rules can get quite verbose. For the sake of brevity, I won't include the `<styles>` element in the rest of the examples in this chapter unless necessary.

8.3.5. Custom elements

The core UI elements that ship with Hyperview are quite basic. Most mobile apps require richer elements to deliver a great user experience. Luckily, HXML can easily accomodate custom elements in its syntax. This is because HXML is really just XML, aka "Extensible Markup Language". Extensibility is already built into the format! Developers are free to define new elements and attributes to represent custom elements.

Let's see this in action with a concrete example. Assume that we want to add a map element to our Hello World app. We want the map to display a defined area, and one or more markers at specific coordinates in that area. Let's translate these requirements into XML:

- An `<area>` element will represent the area displayed by the map. To specify the area, the element will include attributes for `latitude` and `longitude` for the center of the area, and a `latitude-delta` and `longitude-delta` indicating the +/- display area around the center.
- A `<marker>` element will represent a marker in the area. The coordinates of the marker will be defined by `latitude` and `longitude` attributes on the marker.

Using these custom XML elements, an instance of the map in our app might look like this:

```
<doc xmlns="https://hyperview.org/hyperview">
  <screen>
    <body>
      <view>
        <text>Hello World!</text>
        <area latitude="37.8270" longitude="122.4230" latitude-delta="0.1"
longitude=delta="0.1"> ❶
          <marker latitude="37.8118" longitude="-122.4177" /> ❷
        </area>
      </view>
    </body>
  </screen>
</doc>
```

❶ Custom element representing the area rendered by the map

❷ Custom element representing a marker rendered at specific coordinates on the map

The syntax feels right at home among the core HXML elements. However, there's a potential problem. "area" and "marker" are pretty generic names. I could see `<area>` and `<marker>` elements being used by a customization to render charts & graphs. If our app renders both maps and charts, the HXML markup would be ambiguous. What should the client render when it sees `<area>` or `<marker>`?

This is where XML namespaces come in. XML namespaces eliminate ambiguity and collisions between elements and attributes used to represent different things. Remember that the `<doc>` element declares that `https://hyperview.org/hyperview` is the default namespace for the entire document. Since no other elements define namespaces, every element in the example above is part of the `https://hyperview.org/hyperview` namespace.

Let's define a new namespace for our map elements. Since this namespace will not be the default for the document, we also need to assign the namespace to a prefix we will add to our elements:

```
<doc                          xmlns="https://hyperview.org/hyperview"
xmlns:map="https://mycompany.com/hyperview-map">
```

This new attribute declares that the `map:` prefix is associated with the namespace

"https://mycompany.com/hyperview-map". This namespace could be anything, but remember the goal is to use something unique that won't have collisions. Using your company/app domain is a good way to guarantee uniqueness. Now that we have a namespace and prefix, we need to use it for our elements:

```
<doc xmlns="https://hyperview.org/hyperview"
xmlns:map="https://mycompany.com/hyperview-map"> ❶
  <screen>
    <body>
      <view>
        <text>Hello World!</text>
        <map:area latitude="37.8270" longitude="122.4230" latitude-delta="0.1"
longitude=delta="0.1"> ❷
          <map:marker latitude="37.8118" longitude="-122.4177" /> ❸
        </map:area> ❹
      </view>
    </body>
  </screen>
</doc>
```

❶ Definition of namespace aliased to "map"

❷ Adding the namespace to the "area" start tag

❸ Adding the namespace to the "marker" self-closing tag

❹ Adding the nemsapce to the "area" end tag

That's it! If we introduced a custom charting library with "area" and "marker" elements, we would create a unique namespace for those elements as well. Within the HXML doc, we could easily disambiguate `<map:area>` from `<chart:area>`.

At this point you might be wondering, "how does the Hyperview client know to render a map when my doc includes `<map:area>`"? It's true, so far we only defined the custom element format, but we haven't implemented the element as a feature in our app. We will get into the details of implementing custom elements later in the next chapter.

8.3.6. Behaviors

As discussed in earlier chapters, HTML supports two basic types of interactions:

- Clicking a hyperlink: the client will make a GET request and render the response as a

new web page.

- Submitting a form: the client will make a (typically) POST request with the serialized content of the form, and render the response as a new web page.

Clicking hyperlinks and submitting forms is enough to build simple web applications. But relying on just these two interactions limits our ability to build richer UIs. What if we want something to happen when the user mouses over a certain element, or perhaps when they scroll some content into the viewport? We can't do that with basic HTML. Additionally, both clicks and form submits result in loading a full new web page. What if we only want to update a small part of the current page? This is a very common scenario in rich web applications, where users expect to fetch and update content without navigating to a new page.

So with basic HTML, interactions (clicks and submits) are limited and tightly coupled to a single action (loading a new page). Of course, using JavaScript, we can extend HTML and add some new syntax to support our desired interactions. Htmx (and Intercooler before it) do exactly that with a new set of attributes:

- Interactions can be added to any element, not just links and forms.
- The interaction can be triggered via a click, submit, mouseover, or any other JavaScript event.
- The actions resulting from the trigger can modify the current page, not just request a new page.

By decoupling elements, triggers, and actions, htmx allows us to build rich Hypermedia-driven applications in a way that feels very compatible with HTML syntax and server-side web development.

HXML takes the idea of defining interactions via triggers & actions and builds them into the spec. We call these interactions "behaviors". We use a special `<behavior>` element to define then. Here's an example of a simple behavior that pushes a new mobile screen onto the navigation stack:

```
<text>
  <behavior ❶
    trigger="press" ❷
    action="push" ❸
    href="/next-screen" ❹
  />
  Press me!
</text>
```

❶ The element encapsulating an interaction on the parent `<text>` element.

❷ The trigger that will execute the interaction, in this case pressing the `<text>` element.

❸ The action that will execute when triggered, in this case pushing a new screen onto the current stack.

❹ The href to load on the new screen.

Let's break down what's happening in this example. First, we have a `<text>` element with the content "Press me!". We've shown `<text>` elements before in examples of HXML, so this is nothing new. But now, the `<text>` element contains a new child element, `<behavior>`. This `<behavior>` element defines an interaction on the parent `<text>` element. It contains two attributes that are required for any behavior:

- `trigger`: defines the user action that triggers the behavior
- `action`: defines what happens when triggered

In this example, the `trigger` is set to `press`, meaning this interaction will happen when the user presses the `<text>` element. The `action` attribute is set to `push`. `push` is an action that will push a new screen onto the navigation stack. Finally, Hyperview needs to know what content to load on the newly pushed screen. This is where the `href` attribute comes in. Notice we don't need to define the full URL. Much like in HTML, the `href` can be an absolute or relative URL.

So that's a first example of behaviors in HXML. You may be thinking this syntax seems quite verbose. Indeed, pressing elements to navigate to a new screen is one of the most common interactions in a mobile app. It would be nice to have a simpler syntax for the common case. Luckily, `trigger` and `action` attributes have default values of `press` and

`push`, respectively. Therefore, they can be ommitted to clean up the syntax:

```
<text>
  <behavior href="/next-screen" /> ❶
  Press me!
</text>
```

❶ When pressed, this behavior will open a new screen with the given URL.

This markup for the `<behavior>` will produce the same interaction as the earlier example. With the default attributes, the `<behavior>` element looks similar to an anchor `<a>` in HTML. But the full syntax achieves our goals of decoupling elements, triggers, and actions:

- Behaviors can be added to any element, they are not limited to links and forms.
- Behaviors can specify an explicit `trigger`, not just clicks or form submits.
- Behaviors can specify an explicit `action`, not just a request for a new page.
- Extra attributes like `href` provide more context for the action.

Additionally, using a dedicated `<behavior>` element means a single element can define multiple behaviors. This let us execute several actions from the same trigger. Or, we can execute different actions for different triggers on the same element. We will show examples of the power of multiple behaviors at the end of this chapter. First we need to show the variety of supported actions and triggers.

Actions

Behavior actions in Hyperview fall into 3 general categories:

- Navigation actions, which load new screens and move between them
- Update actions, which modify the HXML of the current screen
- System actions, which interact with with OS-level capabilities.

Navigation Actions

We've already seen the simplest type of action, `push`. We classify `push` as a "navigation action", since it's related to navigating screens in the mobile app. Pushing a screen onto the navigation stack is just one of several navigation actions supported in Hyperview. Users also need to be able to go back to previous screens, open and close modals, switch between

tabs, or jump to arbitrary screens. Each of these types of navigations is supported through a different value for the `action` attribute:

- `push`: Push a new screen into the current navigation stack. This looks like a screen sliding in from the right, on top of the current screen.
- `new`: Open a new navigation stack as a modal. This looks like a screen sliding in from the bottom, on top of the current screen.
- `back`: This is a complement to the `push` action. It pops the current screen off of the navigation stack (sliding it to the right).
- `close`: This is a complement to the `new` action. It closes the current navigation stack (sliding it down).
- `reload`: Similar to a browser's "refresh" button, this will re-request the content of the current screen.
- `navigate`: This action will attempt to find a screen with the given `href` already loaded in the app. If the screen exists, the app will jump to that screen. If it doesn't exist, it will act the same as `push`.

`push`, `new`, and `navigate` all load a new screen. Thus, they require an `href` attribute so that Hyperview knows what content to request for the new screen. `back`, `close` do not load new screens, and thus do not require the `href` attribute. `reload` is an interesting case. By default, it will use the URL of the screen when re-requesting the content for the screen. However, if you want to replace the screen with a different one, you can provide an `href` attribute with `reload` on the behavior element.

Let's look at an example that uses several navigation actions on one screen:

```
<screen>
  <body>
    <header>
      <text>
        <behavior action="back" /> ❶
        Back
      </text>

      <text>
        <behavior action="new" href="/widgets/new" /> ❷
        New
      </text>
    </header>
    <text>
      <behavior action="reload" /> ❸
      Check for new widgets
    </text>
    <list>
      <item key="widget1">
        <behavior action="push" href="/widgets/1" /> ❹
      </item>
    </list>
  </body>
</screen>
```

❶ Takes the user to the previous screen

❷ Opens a new modal to add a widget

❸ Reloads the content of the screen, showing new widgets from the backend

❹ Pushes a new screen with details for a specific widget

Update Actions

Behavior actions are not just limited to navigating between screens. They can also be used to change the content on the current screen. We call these "update actions". Much like navigation actions, update actions make a request to the backend. However, the response is not an entire HXML document, but a fragment of HXML. This fragment is added to the HXML of the current screen, resulting in an update to the UI. The `action` attribute of the `<behavior>` determines how the fragment gets incorporated into the HXML. We also need to introduce a new `target` attribute on `<behavior>` to define where the fragment gets incorporated in the existing doc. The `target` attribute is an ID reference to an existing

element on the screen.

Hyperview supports the current update actions, representing different ways to incorporate the fragment into the screen:

- `replace`: replaces the entire target element with the fragment
- `replace-inner`: replaces the children of the target element with the fragment
- `append`: adds the fragment after the last child of the target element
- `prepend`: adds the fragment before the first child of the target element.

Let's look at some examples to make this more concrete. For these examples, let's assume our backend accepts `GET` requests to `/fragment`, and the response is a fragment of HXML that looks like `<text>My fragment</text>`.

```
<screen>
  <body>
    <text>
      <behavior action="replace" href="/fragment" target="area1" /> ❶
      Replace
    </text>
    <view id="area1">
      <text>Existing content</text>
    </view>

    <text>
      <behavior action="replace-inner" href="/fragment" target="area2" /> ❷
      Replace-inner
    </text>
    <view id="area2">
      <text>Existing content</text>
    </view>

    <text>
      <behavior action="append" href="/fragment" target="area3" /> ❸
      Append
    </text>
    <view id="area3">
      <text>Existing content</text>
    </view>

    <text>
      <behavior action="prepend" href="/fragment" target="area4" /> ❹
      Prepend
    </text>
    <view id="area4">
      <text>Existing content</text>
    </view>

  </body>
</screen>
```

❶ Replaces the area1 element with fetched fragment

❷ Replaces the child elements of area2 with fetched fragment

❸ Appends the fetched fragment to area3

❹ Prepends the fetched fragment to area4

In this example, we have a screen with four buttons corresponding to the four update actions: `replace`, `replace-inner`, `append`, `prepend`. Below each button, there's a corresponding `<view>` containing some text. Note that the `id` of each view matches the `target` on the behaviors of the corresponding button.

When the user presses the first button, the Hyperview makes a request for `/fragment`. Next, it looks for the target, ie the element with id "area1". Finally, it replaces the `<view id="area1">` element with the fetched fragment, `<text>My fragment</text>`. The existing view and text contained in that view will be replaced. To the user, it will look like "Existing content" was changed to "My fragment". In the HXML, the element `<view id="area1">` will also be gone.

The second button behaves in a similar way to the first one. However, the `replace-inner` action does not remove the target element from the screen, it only replaces the children. This means the resulting markup will look like `<view id="area2"><text>My fragment</text></view>`.

The third and fourth buttons do not remove any content from the screen. Instead, the fragment will be added either after (in the case of `append`) or before (`prepend`) the children of the target element.

For completeness, let's look at the state of the screen after a user presses all four buttons:

```
<screen>
  <body>
    <text>
      <behavior action="replace" href="/fragment" target="area1" />
      Replace
    </text>
    <text>My fragment</text> ❶

    <text>
      <behavior action="replace-inner" href="/fragment" target="area2" />
      Replace-inner
    </text>
    <view id="area2">
      <text>My fragment</text> ❷
    </view>

    <text>
      <behavior action="append" href="/fragment" target="area3" />
      Append
    </text>
    <view id="area3">
      <text>Existing content</text>
      <text>My fragment</text> ❸
    </view>

    <text>
      <behavior action="prepend" href="/fragment" target="area4" />
      Prepend
    </text>
    <view id="area4">
      <text>My fragment</text> ❹
      <text>Existing content</text>
    </view>

  </body>
</screen>
```

❶ Fragment completely replaced the target using `replace` action

❷ Fragment replaced the children of the target using `replace-inner` action

❸ Fragment added as last child of the target using `append` action

❹ fragment added as the first child of the target using `prepend` action

The examples above show actions making `GET` requests to the backend. But these actions can also make `POST` requests by setting `verb="post"` on the `<behavior>` element. For both `GET` and `POST` requests, the data from the parent `<form>` element will be serialized and included in the request. For `GET` requests, the content will be URL-encoded and added as query params. For `POST` requests, the content will be form-URL encoded and set on the request body. Since they support `POST` and form data, update actions are often used to send data to the backend.

So far, our example of update actions require getting new content from the backend and adding it to the screen. But sometimes we just want to change the state of existing elements. The most common state to change for an element is its visibility. Hyperview has `hide`, `show`, and `toggle` actions that do just that. Like the other update actions, `hide`, `show`, and `toggle` use the `target` attribute to apply the action to an element on the current screen.

```
<screen>
  <body>
    <text>
      <behavior action="hide" target="area" /> ❶
      Hide
    </text>

    <text>
      <behavior action="show" target="area" /> ❷
      Show
    </text>

    <text>
      <behavior action="toggle" target="area" /> ❸
      Toggle
    </text>

    <view id="area"> ❹
      <text>My fragment</text>
    </view>
  </body>
</screen>
```

❶ Hides the element with id "area".

❷ Shows the element with id "area".

❸ Toggles the visibility of the element with id "area".

❹ The element targeted by the actions.

In this example, the three buttons labeled "Hide", "Show", and "Toggle" will modify the display state of the `<view>` with ID "area". Pressing "Hide" multiple times will have no affect once the view is hidden. Likewise, pressing "Show" multiple times will have no affect once the view is showing. Pressing "Toggle" will keep flipping the visibility status of the element between showing and hidden.

Hyperview comes with other actions that modify the existing HXML. We won't cover them in detail, but I'll mention them breifly here:

- `set-value`: this action can set the value of an input element such as `<text-field>`, `<switch>`, `<select-single>`, etc.
- `select-all` and `unselect-all` work with the `<select-multiple>` element to select/deselect all options.

System Actions

Some standard Hyperview actions don't interact with the HXML at all. Instead, they expose functionality provided by the mobile OS. For example, both Android and iOS support a system-level "Share" UI. This UI allows sharing URLs and messages from one app to another app. Hyperview has a `share` action to support this interaction. It involves a custom namespace, and share-specific attributes.

```
<behavior
  xmlns:share="https://instawork.com/hyperview-share" ❶
  trigger="press"
  action="share" ❷
  share:url="https://www.instawork.com" ❸
  share:message="Check out this website!" ❹
/>
```

❶ Defines the namespace for the share action.

❷ The action of this behavior will bring up the share sheet.

❸ URL to be shared

❹ Message to be shared

We've seen XML namespaces when talking about custom elements. Here, we are using a namespace for the `url` and `message` attributes on the `<behavior>`. These attribute names are generic and likely used by other components and behaviors, so the namespace ensures there will be no ambiguity. When pressed, the "share" action will trigger. The values of the `url` and `message` attributes will be passed to the system Share UI. From there, the user will be able to share the URL & message via SMS, email, or other communication apps.

The `share` action shows how a behavior action can use custom attributes to pass along extra data needed for the interactions. But some actions require even more structured data. This can be provided via child elements on the `<behavior>`. Hyperview uses this to implement the `alert` action. The `alert` action shows a customized system-level dialog box. This dialog needs configuration for a title and message, but also for customized buttons. Each button needs to then trigger another behavior when pressed. This level of configuration cannot be done with just attributes, so we use custom child elements to represent the behavior of each button.

```
<behavior
  xmlns:alert="https://hyperview.org/hyperview-alert" ❶
  trigger="press"
  action="alert" ❷
  alert:title="Continue to next screen?" ❸
  alert:message="Are you sure you want to navigate to the next screen?" ❹
>
  <alert:option alert:label="Continue"> ❺
    <behavior action="push" href="/next" /> ❻
  </alert:option>
  <alert:option alert:label="Cancel" /> ❼
</behavior>
```

❶ Defines the namespace for the alert action.

❷ The action of this behavior will bring up a system dialog box.

❸ Title of the dialog box.

❹ A "continue" option in the dialog box

❺ When "continue" is pressed, push a new screen onto the navigation stack.

❻ A "cancel" option that dismisses the dialog box.

Like the `share` behavior, `alert` uses a namespace to define some attributes and elements. The `<behavior>` element itself contains the `title` and `message` attributes for the dialog box. The button options for the dialog are defined using a new `<option>` element nested in the `<behavior>`. Notice that each `<option>` element has a label, and then optionally contains a `<behavior>` itself! This structure of the HXML allows the system dialog to trigger any interaction that can be defined as a `<behavior>`. In the example above, pressing the "Continue" button will open a new screen. But we could just as easily trigger an update action to change the current screen. We could even open a share sheet, or a second dialog box. But please don't do that in a real app! With great power comes great responsibility.

Custom Actions

You can build a lot of mobile UIs with Hyperview's standard navigation, update, and system actions. But the standard set may not cover all interactions you will need for your mobile app. Luckily, the action system is extensible. In the same way you can add custom elements to Hyperview, you can also add custom behavior actions. Custom actions have a similar syntax to the `share` and `alert` actions, using namespaces for attributes that pass along extra data. Custom actions also have full access to the HXML of the current screen, so they can modify the state or add/remove elements from the current screen. In the next chapter, we will create a custom behavior action to enhance our mobile contacts app.

Triggers

We've already seen the simplest type of trigger, a `press` on an element. Hyperview supports many other common triggers used in mobile apps.

`longPress`

Closely related to a press is a long-press. A behavior with `trigger="longPress"` will trigger when the user presses and holds on the element. "Long-press" interactions are often used for shortcuts and power features. Sometimes, elements will support different actions for both a `press` and `longPress`. This is done using multiple `<behavior>` elements on the same UI element.

```
<text>
  <behavior trigger="press" action="push" href="/next-screen" /> ❶
  <behavior trigger="longPress" action="push" href="/secret-screen" /> ❷
  Press (or long-press) me!
</text>
```

❶ Normal press will open the next screen

❷ Long press will open a different screen

In this example, a normal press will open a new screen and request content from `/next-screen`. However, a long press will open a new screen with content from `/secret-screen`. This is a contrived example for the sake of brevity. A better UX would be for the long-press to bring up a contextual menu of shortcuts and advanced options. This could be achieved by using `action="alert"` and opening a system dialog box with the shortcuts.

`load`

Sometimes we want an action to trigger as soon as the screen loads. `trigger="load"` does exactly this. One use case is to quickly load a shell of the screen, and then fill in the main content on the screen with a second update action.

```
<body>
  <view>
    <text>My app</text>
    <view id="container"> ❶
      <behavior trigger="load" action="replace" href="/content" target="container">
❷
      <text>Loading...</text> ❸
    </view>
  </view>
</body>
```

❶ Container element without the actual content

❷ Behavior that immediately fires off a request for /content to replace the container

❸ Loading UI that appears until the content is fetched and replaced.

In this example, We load a screen with a heading ("My app") but no content. Instead, we show a `<view>` with ID "container" and some "Loading…" text. As soon as this screen

loads, the behavior with `trigger="load"` fires off the `replace` action. It requests content from the `/content` path and replaces the container view with the response.

`visible`

Unlike `load`, the `visible` trigger will only execute the behavior when the element with the behavior is scrolled into the viewport on the mobile device. This allows us to The `visible` action is commonly used to implement an infinite-scroll interaction on a `<list>` of `<item>` elements. The last item in the list includes a behavior with `trigger="visible"`. The `append` action will fetch the next page of items and append them to the list.

`refresh`

This trigger captures a "pull to refresh" action on `<list>` and `<view>` items. This interaction is associated with fetching up-to-date content from the backend. Thus, it's typically paired with an update or reload action to show the latest data on the screen.

```
<body>
  <view scroll="true">
    <behavior trigger="refresh" action="reload" /> ❶
    <text>No items yet</text>
  </view>
</body>
```

❶ When the view is pulled down to refresh, reload the screen

Note that adding a behavior with `trigger="refresh"` to a `<view>` or `<list>` will add the pull-to-refresh interaction to the element, including showing a spinner as the element is pulled down.

`focus`, `blur`, *and* `change`

These triggers are related to interactions with input elements. Thus, they will only trigger behaviors attached to elements like `<text-field>`. `focus` and `blur` will trigger when the user focuses and blurs the input element, respectively. `change` will trigger when the value of the input element changes, like when the user types a letter in a text field. These triggers are often used with behaviors that need to perform some server-side validation on the form fields. For example, when the user types in a username and then blurs the field, a behavior could trigger on `blur` to make a request to the backend and check for uniqueness

of the username. If the entered username is not unique, the response could include an error message letting the user know they need to pick a different username.

Using Multiple Behaviors

Most of the example shown above attach a single `<behavior>` to an element. But there's no such limitation in Hyperview; elements can define multiple behaviors. We already saw an example where a single element had different actions triggered on `press` and `longPress`. But we can also trigger multiple actions on the same trigger.

```
<screen>
  <body>
    <text id="area1">Area 1</text>

    <text>
      <behavior action="hide" target="area1" /> ❶
      <behavior action="hide" target="area2" /> ❷
      Hide
    </text>

    <text id="area2">Area 2</text>
  </body>
</screen>
```

❶ Hide element with ID "area1" when pressed

❷ Hide element with ID "area2" when pressed

In this admittedly contrived example, we want to hide 2 elements on the screen when pressing the "Hide" button. The two elements are far apart in the HXML, and cannot be hidden by hiding a common parent element. But, we can trigger two behaviors at the same time, each one executing a "hide" action but targeting different elements.

8.4. Summary

At the beginning of this chapter, I made a case for developing mobile apps using a Hypermedia architecture. I also explained why using HTML & web views is not the best approach to deliver a native-feeling mobile experience. Hyperview as a framework allow us to develop native mobile apps without giving up the Hypermedia architecture. Unlike HTML, HXML is a mobile-first format with first-class support for the interactions and patterns users expect on mobile. HXML has built-in rich Hypermedia functionality inspired

by htmx and IntercoolerJS. However, HXML takes things a step further with the concept of behaviors and custom elements/actions. By controlling the Hyperview client, developers are free to extend HXML to suit their needs. All this can be done without writing and shipping scripts embedded in HXML. In fact, the concept of scripting doesn't exist in the format! This allows HXML to remain pure and declarative, while the client evolves new capabilites over time.

In the next chapter, we will put the ideas of Hyperview into action, by porting the Contacts App from a web app to a native mobile application. = Hypermedia In Action :chapter: 8 :sectnums: :figure-caption: Figure 6. :listing-caption: Listing 6. :table-caption: Table 6. :sectnumoffset: 7 :leveloffset: 1 :sourcedir: ../code/src :source-language:

9. Client Side Scripting

This chapter covers

- How scripting can be effectively added to a Hypermedia Driven Application
- Adding a javascript-based confirmation dialog for deleting contacts
- Adding a three-dot menu in our contacts table
- Adding a keyboard shortcut for focusing the search input
- Adding support for re-ordering contacts via drag-and-drop

Scripting in Hypermedia-Driven Applications

> REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.
>
> — Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

Thus far we have avoided writing any JavaScript for Contact.app, mainly because the functionality we implemented so far does not need it. Contrary to popular belief, hypermedia is not just for "documents" (where a document is considered essentially different to an "app"), and it has many affordances for building interactive experiences. We want to show that it is possible to build sophisticated web applications using the original model of the web without the abstractions provided by JavaScript frameworks. On the other hand, htmx itself is written in JavaScript, and we don't want our message to be interpreted

as "JavaScript bad", or, more generally, "Client-side scripting bad."

[htmx loves javascript] | *htmx-loves-javascript.png*

So the question isn't "Should we be scripting for the web?" but rather "How should we be scripting for the web?"

Scripting has been a massive multiplier of the Web's capabilities. Through its use, Web application authors are not only able to enhance their hypertext-based websites, but also create full-fledged client-side applications that can compete with native apps in how they work (although they don't always win when they do). In other terms, the Web became a distribution medium for non-REST apps in addition to being a RESTful system. When it's not used as a replacement for the RESTful architecture provided by the Web, however, scripting is extremely useful in Hypermedia Driven Applications.

You are scripting in a way compatible with HDAs if:

- The main data format exchanged between server of client is hypermedia, the same as it would be in an application with no scripting.
- Client-side state (other than the DOM) is minimized.

This style of scripting requires us to different practices than what is usually recommended for JavaScript, as the most common advice often comes from SPA or server-side backgrounds. We will see these new practices in action in the upcoming chapter.

However, listing "best practices" is rarely convincing or edifying (and often boring). So, we instead frame them around shiny tools that work well for scripting in a HDA. We will use each of these tools to add a feature to ContactApp:

- An overflow menu to hold the *Edit*, *View* and *Delete* actions, to clean up visual clutter in our list of contacts
- Reordering contacts by dragging and dropping
- A dialog to confirm the deletion of contacts
- A keyboard shortcut for focusing the search box

The important idea in the implementation of each of these features is that they are entirely client-side and don't exchange information with the server using, for example, JSON. This

is what will keep them all within the bounds of a proper Hypermedia Driven Application.

9.1. Scripting tools for the Web

The primary scripting language for the web is, of course, JavaScript, which is ubiquitous in web development today. A bit of interesting internet lore, however, is that JavaScript was not always the only built-in option. As the quote from Roy Fielding above indicates, *applets* written in other languages such as Java were considered part of the scripting infrastructure of the web. In addition, there was a brief period when Internet Explorer supported VBScript, a scripting language based on Visual Basic.

Today, we have a variety of *transcompilers* (often shortened to *transpilers*) that convert another language to JavaScript, such as TypeScript, Dart, Kotlin, ClojureScript, F#. There is also the WebAssembly bytecode format, which is supported as a compilation target for C, Rust, and the WASM-first language AssemblyScript. However, most of these are not geared towards an HDA-compatible style of scripting --- compile-to-JS languages are often paired with SPA-oriented libraries (Dart and AngularDart, ClojureScript and Reagent, F# and Elmish), and WASM is currently mainly geared toward linking to C/C++ libraries from JavaScript.

I bring this up because we are going to look at three different mechanisms for adding scripting to our Hypermedia Driven Application:

- Vanilla JS, that is, using JavaScript without depending on any framework.
- Alpine.js, a JavaScript library for adding behavior directly in HTML.
- _hyperscript, a non-JavaScript scripting language created alongside htmx. Like AlpineJS, it is usually embedded in HTML.

Let’s take a quick look at each of these scripting options so we know what we are dealing with. As with CSS, we are not going to deep dive into any of these options: we are going to show just enough to give you a flavor of each and, we hope, spark your interest in looking into each of them more extensively.

9.2. Vanilla JavaScript

> No code is faster than no code.
>
> — Merb

Vanilla JavaScript is simply using JavaScript in your application without any intermediate layers. The term came into vogue as a play on the fact that there were so many ".js" frameworks out there to help you write JavaScript. As JavaScript matured as a scripting language, standardized across browsers and provided more and more functionality, the utility of many of these frameworks and libraries has diminished. (At the same time, however, SPAs became more popular, requiring better frameworks).

A quote from the humorous website http://vanilla-js.com captures the situation well:

> Vanilla JS is the lowest-overhead, most comprehensive framework I've ever used.
>
> — http://vanilla-js.com

The message of *VanillaJS* here is that since the browser already has JavaScript baked into it, there isn't any need to download a framework for your application to function. This is true more often than we like to admit and especially so in HDAs as hypermedia obviates many features provided by frameworks:

- Client-side routing
- An abstraction over DOM manipulation, i.e.: templates that automatically update when referenced variables change
- Server side rendering (rendering here refers to HTML generation)
- Attaching dynamic behavior to server-rendered tags on load
- Network requests

Installation of VanillaJS couldn't be easier: you don't have to. You can just start writing JavaScript in your web application and it will simply work.

That's the good news. The bad news is that JavaScript has some limitations as a scripting language that often make it less than ideal as a stand-alone scripting technology for Hypermedia Driven Applications:

- It is a relatively complex language that has accreted a lot of features and warts.
- Its model for asynchrony involves *colored functions*, a concept described in Robert Nystrom's oft-cited *What Color is Your Function?* [1]
- It is surprisingly clunky to work with events in the language.

- DOM APIs (a large portion of which were originally designed for Java) are verbose and do not make common functionality easy to use.

None of these are deal breakers, of course, and many people prefer the "close to the metal" (for lack of a better term) nature of vanilla JavaScript to more elaborate client-side scripting approaches.

As our "hello world" example to showcase each of our scripting options, let's write a counter [2]. It will have a number and a button that increments the number. Nothing too elaborate, but it will give you the flavor of each of the three scripting approaches we are going to use in this chapter.

We have a problem, however, as one of the things frameworks provide is still missing: a standardized code style. This is not an insurmountable problem, and in fact a great opportunity to take a small journey through various styles, starting with the simplest thing possible.

Listing 6. 95. Counter in vanilla JavaScript, inline version

```
<section class="counter">
  <output id="my-output">0</output> ❶
  <button
    onclick=" ❷
      document.querySelector('#my-output') ❸
        .textContent++ ❹
    "
  >Increment</button>
</section>
```

❶ Our output element has an ID to help us find it

❷ We use the `onclick` attribute, a brittle but quick way to add an event listener

❸ Find the output

❹ JavaScript lets us use the `++` operator on a string because it loves us

So, not too bad. It's a little annoying that we needed to add an `id` to the span to make this work and `document.querySelector` is a bit verbose compared to, say, `$` but (but!) it works and it doesn't require any other JavaScript libraries.

A more "standard" way to write the above would be to put the above in a separate file,

either linked via a `<script src>` or placed into an inline `<script>` by a build process:

Counter in vanilla JavaScript, in multiple files

```
<section class="counter">
  <output id="my-output">0</output>
  <button class="increment-btn">Increment</button>
</section>
```

```
const counterOutput = document.querySelector("#my-output") ❶
const incrementBtn  = document.querySelector(".counter .increment-btn") ❷

incrementBtn.addEventListener("click", e => { ❸
  counterOutput.innerHTML++ ❹
})
```

❶ Find the output element

❷ and the button

❸ We use `addEventListener`, which is preferable to `onclick` for many reasons

❹ The logic stays the same, only the structure around it changes

The main reason people do this is for the sake of *separation of concerns.*

The purpose of separating concerns is that we will be able to modify one with confidence that we won't break any other. Is this really the case with HTML and JS?

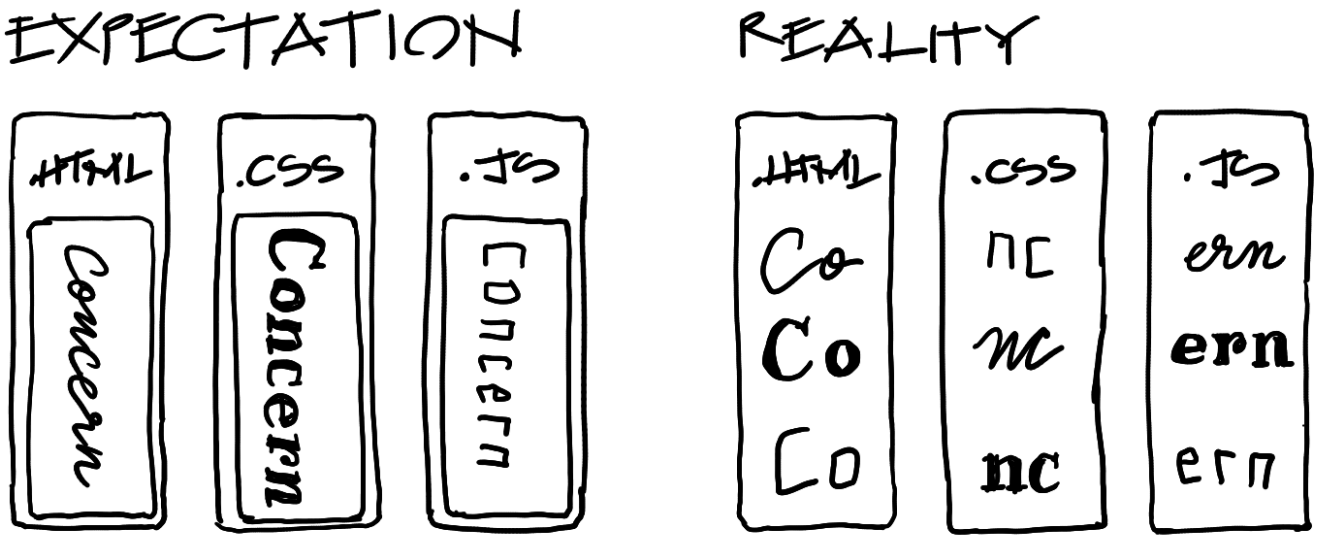
Notice that the HTML in the above example is not just the previous example with the onclick attribute removed. Can you spot the difference?

We've had to add a class to the button so that we could find it in JS. In both the HTML and the JS, this class is a string literal not subject to name resolution (the process, in compilers and interpreters, of linking names to what they reference). The careless use of CSS selectors in JavaScript causes *jQuery soup*, where:

- The JS that attaches behavior to a given element is unclear (though developer tools in browsers help with this).
- Reuse is difficult.

- The code is disorganized (if we have many components, how do we separate them into files (if at all?))

Furthermore, imagine that we want to change the number field from an `<output>` tag to an `<input type="number">`. This change to our HTML will break our JavaScript. The fix is trivial (change `.textContent` to `.value`), but it's not hard to see how the burden of synchronizing markup and code across files would increase in larger components or across a whole page.



The tight coupling between files in this simple example suggests that separation between HTML and JavaScript (and CSS) is often an illusory separation of concerns. Contact.app is is not *concerned* with "structure", "styling" or "behavior", it's concerned with collecting contact info and presenting it.

Our suspicion is validated by developments in the JS framework world:

- JSX
- LitHTML
- CSS-in-JS
- Single-File Components
- Filesystem based routing

All of these solutions colocate code in various languages that address a single feature

(usually, a UI widget). In order to use them effectively, we need to understand the problem domain and identify business concerns in adition to implementation concerns.

9.2.1. Locality of Behavior

Locality of Behavior (LoB) is a software design principle that we coined to describe the following characteristic of a piece of software:

> The behaviour of a unit of code should be as obvious as possible by looking only at that unit of code.
>
> — https://htmx.org/essays/locality-of-behaviour/

In simple terms: you should be able to tell what a button does by simply looking at the code or markup that creates that button. This does not mean you need to inline the entire implementation, but that you shouldn't need to hunt for it or require prior knowledge of the codebase to find it.

We will demonstrate Locality of Behavior in all of our examples, both the counter demos and the features we add to ContactApp. It is a design goal of both _hyperscript and Alpine.js (which we will cover later) as well as htmx. These tools achieve it through having you embed attributes and directly within your HTML, as opposed to having code pluck elements out of a document through CSS selectors and add event listeners onto them.

The `addEventListener` method is, in a way, monkey-patching. Its functionality is the same for event listeners as ruby's `define_method`:

Listing 6. 96. `define_method` **in Ruby**

```
button.define_method(:click, ->{ ❶
  count += 1 ❷
})
```

❶ When a `click` method call is received,

❷ Do this

Listing 6. 97. `addEventListener` **in JavaScript**

```
button.addEventListener('click', () => { ❶
  count++ ❷
})
```

❶ When a `click` event is received,

❷ Do this

(The Ruby code is deliberately unidiomatic to make it easier to understand for non-Rubyists).

Monkey-patching actually used to be the default way of adding methods in JavaScript. After classes were added in ES2015, however, modifying the `prototype` of a function (which, confusingly, is not the function's prototype but of objects the function returns) is increasingly discouraged. No such advancement has been made for event listeners, however, leaving us stuck with `addEventListener` and `onclick`.

```
'use strict'; ❶
(function () {
  Button.prototype.click = function () {
    count++;
  }
})();
```

❶ Feeling nostalgic yet?

This is a shame, because in the case of front end scripting in a HDA, locality of behavior is often the more important principle over separation of concerns.

2 > 1 > 2

Having two decoupled modules is better than having one big blob, but two tightly-coupled modules is worse than either.

(Of course, having no code at all is the best, so 0 > 2 > 1 > 2.)

So, should we go back to the `onclick` way of doing things? It certainly wins in the Locality of Behavior category. Unfortunately, JavaScript in `on*` attributes are not a great way to program:

- They don't support custom events.
- There is no good mechanism for associating long-lasting variables with an element --- all variables are discarded when an event listener completes executing.

- If you have multiple instances of an element, you will need to repeat the listener code on each, or use something more clever like event delegation.
- JavaScript code that directly manipulates the DOM gets verbose, and clutters the markup.
- An element cannot listen for events on another element. For example, if you want to dismiss a popup by clicking outside it, the listener will need to be on the body element. The body element will need to have listeners that deal with many unrelated components, some of which may not even be on the page if it was generated from a common template.

JavaScript and Locality of Behavior don't seem to mesh as well as we want them to, but the situation is not hopeless. it's important to be aware that LoB does not require behavior to be *defined* at the use site, but merely invoked there. Keeping this in mind, it's possible to improve LoB while writing JS in a separate file, provided we have a reasonable system for structuring our JavaScript.

9.2.2. RSJS

RSJS ("Reasonable System for JavaScript Structure", https://ricostacruz.com/rsjs/) is a set of guidelines for JavaScript architecture targeted at "a typical non-SPA website". RSJS is a solution to the lack of a standard code style we mentioned earlier.

We won't replicate all of the guidelines here, but here are the ones most relevant to this book:

- "Use `data-` attributes" --- invoking behavior via adding data attributes makes it obvious there is JavaScript happening, as opposed to random classes or IDs that may be mistakenly removed or changed
- "One component per file" --- the name of the file should match the data attribute so that it can be found easily, a win for LoB

Counter in vanilla JavaScript, with RSJS

```
<section class="counter" data-counter> ❶
  <output id="my-output" data-counter-output>0</output> ❷
  <button class="increment-btn" data-counter-increment>Increment</button>
</section>
```

❶ Invoke a JavaScript behavior with a data attribute

❷ Mark relevant child elements

```
// counter.js ❶
document.querySelectorAll("[data-counter]") ❷
  .forEach(el => {
    const output = el.querySelector("[data-counter-output]"),
      increment = el.querySelector("[data-counter-increment]") ❸

    increment.addEventListener("click", e => output.textContent++) ❹
  })
```

❶ File should have the same name as the data attribute, so that we can locate it easily

❷ Get all elements that invoke this behavior

❸ Get any child elements we need

❹ Register event handlers

This methodology solves (or at least alleviates) many of our gripes with the previous example of vanilla JS in a separate file:

- The JS that attaches behavior to a given element is **clear** (though only through naming conventions).
- Reuse is **easy** --- you can create another counter on the page and it will just work.
- The code is **well-organized** --- one behavior per file

You may remember the problem we discussed about replacing the output tag with `<input type="number">`. That problem still remains. There is a way to solve it, but it's a bit convoluted:

Counter with vanilla JavaScript, with extra-flexible RSJS

```
<section class="counter" data-counter>
  <output id="my-output" data-counter-output="innerHTML">0</output> ❶
  <button class="increment-btn" data-counter-increment>Increment</button>
</section>
```

❶ Specify the property to put the value into

```
// counter.js
document.querySelectorAll("[data-counter]").forEach(el => {
  const output = el.querySelector("[data-counter-output]"),
    increment = el.querySelector("[data-counter-increment]")

  const outProp = output.dataset.counterOutput ❶

  increment.addEventListener("click", e => output[outProp]++) ❷
})
```

❶ Get the attribute's value

❷ Dynamically access the property to increment

If we wanted to use an input, we would change the value of `data-counter-output` to `"value"`. This would also work with `<input type="range">`!

On one hand, this is a way overengineered the solution to the problem. How often do we need to reuse a counter?

On the other, let's think about where else we could go with this. With very little work, we could let the button markup specify the increment amount --- we could go 5-at-a-time, or decrement (increment by -1). It might be a little more puzzling to support multiple increment buttons with varying amounts if you aren't familiar with this kind of programming, but not insurmountable. As you continue hacking on this counter example, you could end up building a DSL for smart number inputs. The decoupling that is forced on us by putting our JavaScript in a separate file can lead us to invention; restriction breeds creativity.

That's enough fun, however, let's get to work on ContactApp.

Event delegation

Event delegation is a technique that makes use of bubbling in DOM events both as a form of code organization and to reduce memory usage, in situations where a large number of elements need to respond to an event in the same way. Instead of attaching event listeners to each individual element, we attach a single listener to a shared parent element. The parent listener determines which element the event arrived through.

The following is how event delegation would be usually implemented:

Listing 6. 98. With event delegation

```
ul.addEventListener('click', e => {
    const li = e.target.closest('li')
    if (!li) return

    doThingWith(li)
})
```

whereas the alternative would be:

Listing 6. 99. Without event delegation

```
ul.querySelector('li').forEach(li => {
  li.addEventListener('click', e => {
    doThingWith(li)
  })
})
```

Benefits of event delegation

- If elements are dynamically added, there is no need to add the event listener onto them (this usually requires extracting the listener to a named function, and code repeated in every place where events are added). Event delegation can simplify code quite a lot.
- Having only one event listener reduces memory use.
- When code is inline in HTML, event delegation protects us from repetition.

Drawbacks of event delegation

- The listener will execute for every click in a subtree (or other event type) when not all may be relevant.
- The listener will stay around even if no relevant elements remain.

9.2.3. Vanilla JavaScript in action: A confirmation dialog

Right now, clicking the `Delete` link on a contact instantly deletes it, making it prone to accidents. We'll write some JavaScript to add confirmation dialogs to elements and use it on the delete button.

We'll write the JavaScript first before adding anything to our markup.

Listing 6. 100. Confirmation dialog with Vanilla JS & RSJS

```
document.querySelectorAll("[data-confirm]") ❶
  .forEach(el => {
    // ...
  })
```

❶ Find relevant elements. Our attribute is `data-confirm`, so we'll write this code in a file named `confirm.js`.

We need to show a confirmation dialog. There are libraries that let us show styled, rich alert dialogs, but let's just use `confirm()` for now. Adding in a library later will be a good test of how maintainable our code is.

```
document.querySelectorAll("[data-confirm]")
  .forEach(el => {
    el.addEventListener("...", e => { ❶
      const didConfirm = confirm()
      if (!didConfirm) {
        event.stopImmediatePropagation(); ❷
        event.stopPropagation(); ❸
      }
    })
  })
```

❶ **What event?**

❷ Prevent listeners on this element from running

❸ Prevent listeners on parent elements from running

We need to decide what event we need to listen to:

- Hardcode `"click"`. It's simple and it covers most cases. However, there's not a clear escape hatch if you need a different event.
- Try to sniff what event you need to listen to based on the element. Complex and fragile (but I repeat myself).
- Let the author specify in the attribute. This is what we'll do.

```
el.addEventListener(
  el.dataset.confirm || "click", ❶
  e => {
    // ...
  }
)
```

❶ Specify a default for convenience.

In 9 lines of code, we have a generic confirmation library that we can use for any element as follows. It's definitely overengineered as a result of the forced decoupling, just like the counter earlier, but it works well and was reasonably fun to write.

```
<button type="submit" data-confirm>Delete</button>
<input type="radio" name="volume" value="100" data-confirm="input">
```

Async ruins everything

In the confirmation dialog code we wrote, we use `confirm()`, which is convenient, but displays a barebones dialog that cannot contain rich text. Can we write a similar script using a fancy alert dialog library, like SweetAlert2?

```
document.querySelectorAll("[data-confirm]")
  .forEach(el => {
    el.addEventListener("click", e => {
      const result = await Swal.fire("Are you sure?", "", "question")
      const didConfirm = result.isConfirmed
      if (!didConfirm) {
        event.stopImmediatePropagation();
        event.stopPropagation();
      }
    })
  })
```

```
Uncaught SyntaxError: await is only valid in async functions, async generators
and modules
```

Right. Let's fix that…

```
document.querySelectorAll("[data-confirm]")
  .forEach(el => {
    el.addEventListener("click", async e => {
      const result = await Swal.fire("Are you sure?", "", "question")
      const didConfirm = result.isConfirmed
      if (!didConfirm) {
        event.stopImmediatePropagation();
        event.stopPropagation();
      }
    })
  })
```

No more errors, but this code no longer works. This is because by the time we call `stopPropagation` and `stopImmediatePropagation`, the event has already propagated. We can avoid this when using the built-in `confirm` function because it has the privilege of blocking the main thread.

There is no general solution to this problem.

9.3. Alpine.js

Alpine.js (https://alpinejs.dev) is a relatively new JavaScript library that allows you to embed your code directly in HTML. It bills itself as a modern replacement for jQuery, a widely used but quite old JavaScript library, and it lives up to that promise.

Installing AlpineJS is a breeze, you can simply include it via a CDN:

Listing 6. 101. Installing AlpineJS

```
<script src="https://unpkg.com/alpinejs"></script>
```

You can also install it from npm, or vendor it from your own server.

The main interface of Alpine is a set of HTML attributes, the main one of which is `x-data`. The content of `x-data` is a JavaScript expression which evaluates to an object, whose properties we can access in the element. For our counter, the only state is the current number, so let's create an object with one property:

Listing 6. 102. Counter with Alpine, line 1

```
<div class="counter" x-data="{ count: 0 }">
```

We've defined our state, let's actually use it:

Listing 6. 103. Counter with Alpine, lines 1-2

```
<div class="counter" x-data="{ count: 0 }">
  <output x-text="count"></output> ❶
```

❶ The `x-text` attribute.

This attribute sets the text content of an element to a given expression. Notice that we can access the data of a parent element.

To attach event listeners, we use `x-on`:

Listing 6. 104. Counter with Alpine, the full thing

```
<div class="counter" x-data="{ count: 0 }">
  <output x-text="count"></output>

  <button x-on:click="count++">Increment</button> ❶
</div>
```

❶ With `x-on`, we specify the attribute in the attribute *name*.

Would you look at that, we're done already! (It's almost as though we wrote a trivial example). What we created is, incidentally, nearly identical to the second code example in Alpine's documentation --- available at https://alpinejs.dev/start-here.

9.3.1. `x-on:click` *vs.* `onclick`

The `x-on:click` attribute (or its shorthand `@click`) differs from the browser built-in `onclick` attribute in significant ways that make it much more useful:

- You can listen for events from other elements. For example, the `.outside` modifier lets you listen to any click event that is **not** within the element.
- You can use other modifiers to
  - throttle or debounce event listeners,
  - ignore events that are bubbled up from descendant elements, or
  - attach passive listeners.
- You can listen to custom events, such as those dispatched by htmx.

9.3.2. Reactivity and templating

As you can see, this code is much tighter than the VanillaJS implementation. It helps that AlpineJS supports a notion of variables, allowing you to bind the visibility of the `span` element to a variable that both it and the button can access. Alpine allows for much more elaborate data bindings as well, it is an excellent general purpose client-side scripting library.

9.3.3. Alpine in action: an overflow menu

An overflow menu only has one bit of state: whether it is open.

```
<div x-data="{ open: false }"> ❶
  <button>Options</button> ❷
  <div>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
  </div>
</div>
```

❶ Define the initial state

❷ We'll hook this button up to open and close our menu

While we have only one bit of state, we have many parts that depend on it. This is where *reactivity* shines:

```
<div x-data="{ open: false }">
  <button
    aria-haspopup="menu" ❶
    aria-controls="contents" ❷
    x-bind:aria-expanded="open" ❸
  >Options</button>
  <template x-if="open"> ❹
    <div id="contents"> ❺
      <a href="/contacts/{{ contact.id }}/edit">Edit</a>
      <a href="/contacts/{{ contact.id }}">View</a>
    </div>
  </template>
</div>
```

❶ Declare that this button will cause a menu to open,

❷ and that the menu that this button *controls* is the one with ID `contents`

❸ Indicate the current open state of the menu, using x-bind to reference our data

❹ Only show the menu if it is open

❺ Add an ID to the menu, so that we can reference it in the aria-controls attribute

This is based on the Menu Button example from the cite:[ARIA Authoring Practices Guide]. We haven't made the menu work yet, just the button that opens it.

The use `x-bind` means that as we change the open state, the `aria-expanded` attribute

will update to match. The same holds for the `x-show` on the div with the contents, and indeed for most of Alpine. In order to see this in action, let's actually change that state:

HTML ID Soup

Some features of HTML such as ARIA require you to assign unique IDs to elements. When pages are generated from templates dynamically, avoiding name conflicts in large apps can be difficult, as HTML IDs are not scoped the way identifiers in programming languages are.

Some developers in the SPA world use randomized IDs with a tool like https://npmjs.com/nanoid to avoid the issue.

```
<div x-data="{ open: false }">
  <button
    aria-haspopup="menu"
    aria-controls="contents"
    x-bind:aria-expanded="open"
    x-on:click="open = !open" ❶
  >Options</button>
  <template x-if="open">
    <div id="contents" x-on:click.outside="open = false"> ❷
      <a href="/contacts/{{ contact.id }}/edit">Edit</a>
      <a href="/contacts/{{ contact.id }}">View</a>
    </div>
  </template>
</div>
```

❶ Toggle the open state when the button is clicked

❷ Dismiss the menu by clicking away

You should be able to open the menu now, and may be tempted to ship this code to production. Don't! We're not done because our menu fails many requirements for menu interactions:

- It does not have the `menu` or `menuitem` roles applied properly, which makes life harder for users of assistive software
- You can't navigate between menu items using arrow keys

- You can't activate a menu item with the Space key

These factors make our menu annoying and even unusable for many people. Let's fix it with the guidance of the venerable cite:[ARIA Authoring Practices Guide]:

```
<div x-data="{ open: false }">
  <button
    aria-haspopup="menu"
    aria-controls="contents"
    x-bind:aria-expanded="open"
    x-on:click="open = !open"
  >Options</button>
  <div role="menu" ❶
    id="contents" x-show="open"
    x-on:click.outside="open = false"
    x-on:keydown.up="document.activeElement.previousElementSibling?.focus()" ❷
    x-on:keydown.down="document.activeElement.nextElementSibling?.focus()" ❸
    x-on:keydown.space="document.activeElement.click()" ❹
    x-effect="if (open) requestAnimationFrame(() => $el.firstElementChild.focus())"
❺❻
    x-on:keydown="$event.key === 'Home'
      ? $el.firstElementChild.focus()
      : $event.key === 'End'
      ? $el.lastElementChild.focus()
      : null" ❼
    >
    <a role="menuitem" ❽
      tabindex="-1" ❾
      href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a role="menuitem" tabindex="-1" href="/contacts/{{ contact.id }}">View</a>
  </div>
</div>
```

❶ Put the `menu` role on the menu root

❷ Move focus to the previous element when the up arrow key is pressed

❸ Move focus to the next element when the down arrow key is pressed

❹ Click the currently focused element when the space key is pressed

❺ Access the div itself through the Alpine-supplied `$el` variable

❻ Focus the first item when `show` changes

❼ Handle the remaining cases that Alpine doesn't have modifiers for

❽ Put the `menuitem` role on the individual items

❾ Make the menu items non-tabbable

`x-effect` is a cool attribute that lets you perform side-effects when a piece of element state changes. It automatically detects which state is accessed in the effect. However, it can also complicate our code --- in this example, we need to use `requestAnimationFrame` because otherwise, the effect is executed before the `x-show` attribute reveals the element to focus.

I'm pretty sure that covers all our bases. That's a lot of code! But it's code that encodes a lot of behavior. Not to mention that we still made some assumptions to make our code shorter:

- All children are menu items with no wrappers, dividers, etc.
- There are no submenus

As we need more features, it might make more sense to use a library --- for instance, GitHub's `details-menu-element`.

9.3.4. Reusable behavior in Alpine

Our menu component has a lot of attributes that will currently be repeated in every item of the table. This is hard to maintain when manually writing HTML and increases payload sizes when generating it via a template. We can rectify this using an nifty feature of the `x-bind` attribute:

> x-bind allows you to bind an object of different directives and attributes to an element.
>
> The object keys can be anything you would normally write as an attribute name in Alpine. This includes Alpine directives and modifiers, but also plain HTML attributes. The object values are either plain strings, or in the case of dynamic Alpine directives, callbacks to be evaluated by Alpine.
>
> — https://alpinejs.dev/directives/bind#bind-directives

It's far easier to understand what this means after seeing the attribute in use. To begin, we create a JavaScript function which will encapsulate all of our menu's behavior:

```
function menu() {
  return {
    role: "menu",
    "x-show"() { ❶
      return this.open; ❷
    },
    "x-on:click.outside"() { this.open = false },
    "x-on:keydown.up"() { document.activeElement.previousElementSibling?.focus() },
    "x-on:keydown.down"() { document.activeElement.nextElementSibling?.focus() },
    "x-on:keydown.space"() { document.activeElement.click() },
    "x-effect"() { if (this.open) this.$el.firstElementChild.focus() },
    "x-on:keydown"(event) { ❸
      if (event.key === 'Home') $el.firstChild.focus()
      else if (event.key === 'End') $el.lastChild.focus()
    },
  }
}
```

❶ JavaScript allows any string literal to be the name of an object member. This even works with classes!

❷ Values that would be globally accessible in an attribute are accessed through `this` in a function.

❸ We can clean up longer functions.

The return value is a map of attribute names to values, with Alpine attributes having functions as values instead of strings of code. We can then reference this function in HTML as follows:

```
<div id="contents" x-bind="menu()">
  <a role="menuitem" tabindex="-1" href="/contacts/{{ contact.id }}/edit">Edit</a>
  <a role="menuitem" tabindex="-1" href="/contacts/{{ contact.id }}">View</a>
</div>
```

This requires the function `menu` to be global. We can avoid that with `Alpine.data`, which is a function to make any data accessible to Alpine expressions:

```
Alpine.data("menu", () => {
  return {
    role: "menu",
    "x-show"() { return this.open; },
    // ...
  }
})
```

Another useful tool in factoring Alpine code is calling functions in `x-data` as follows:

```
Alpine.data("toggleableMenu", () => ({ open: false }))
```

```
<div x-data="toggleableMenu()">
  <button
    aria-haspopup="menu"
    ...>
```

You can combine the two techniques:

```
Alpine.data("toggleableMenu", () => ({
  open: false,
  menuBehavior: { ❶
    role: "menu",
    "x-show"() { return this.open; },
    // ...
  },
  buttonBehavior: { ❷
    "aria-haspopup": "menu",
    "aria-controls": "contents",
    "x-bind:aria-expanded"() { return this.open },
    "x-on:click"() { this.open = !this.open },
  }
}))
```

❶ The object that we bind to the menu has been moved into the data.

❷ We can encapsulate the button's behavior in the same way.

```
<div x-data="toggleableMenu()">
  <button x-bind="buttonBehavior">Options</button> ❶
  <div id="contents" x-bind="menuBehavior"> ❷
    <!-- ... -->
```

❶ Access the button behavior object from the data.

❷ Same for the menu… hey, does this look familiar?

You may notice that the markup for the `x-bind` style quite resembles RSJS. Combined with Alpine's reactivity and concise syntax, it's quite a powerful style for writing localized as well as decoupled code.

Factoring our behavior in this way reduces the locality in our code, as it requires us to locate the `menu` and `toggleableMenu` functions to understand what our code does. You can use named files similarly to RSJS to somewhat alleviate this issue, but it's a tradeoff that needs to be considered.

9.4. _hyperscript

While previous two examples are JavaScript-oriented, _hyperscript (https://hyperscript.org, the underscore is part of the name but not pronounced) is a entire new scripting language for front-end development. It has a completely different syntax than JavaScript, derived from an older language called HyperTalk, which was the scripting language of HyperCard, an old hypermedia system, along with IDE and WYSIWYG editor on the Macintosh Computer. The most noticeable thing about _hyperscript is that it resembles English prose more than it does code. It was initially created as a sister project to htmx, to handle events and modify the document in htmx-based applications. Currently, it positions itself as a modern jQuery replacement and alternative to JavaScript.

Like Alpine, _hyperscript allows you to program inline in HTML, but instead of using JavaScript, it has a syntax designed to be embedded into other languages.

What it eschews is a reactive mechanism, instead focusing on making manual DOM manipulation easier. It has built-in constructs for many DOM operations, preventing you from needing to navigate sometimes-verbose APIs.

We will not be doing a deep dive on the language, but again just want to give you a flavor of what scripting in _hyperscript is like, so you can pursue the language in more depth later

if you find it interesting.

Like htmx and AlpineJS, _hyperscript can be installed via a CDN or from npm (package name `hyperscript.org`):

Listing 6. 105. Installing _hyperscript via CDN

```
<script src="//unpkg.com/hyperscript.org"></script>
```

Like AlpineJS, in _hyperscript you put attributes directly in your HTML. Unlike AlpineJS, there is only one attribute for _hyperscript: the `_` (underscore) attribute [3]. This is where all the code responsible for an element goes.

```
<div class="counter">
  <output>0</output>
  <button _="on click increment the textContent of the previous <output
/>">Increment</button> ❶
</div>
```

❶ This is what _hyperscript looks like, believe it or not!

Seasoned JavaScript programmers are often suspicious of _hyperscript: There have been many "natural language programming" projects that usually target non-programmers and beginner programmers, assuming that being able to read code will give you the ability to write it as well. (The authors' views on the usefulness of natural language for teaching programming are nuanced and out of scope for this book). It should be noted that _hyperscript is openly a programming language, in fact, its syntax is inspired in many places by the speech patterns of web developers. In addition, _hyperscript's readability is achieved not through complex heuristics or NLP, but common parsing tricks and a culture of readability.

As you can see in the above example, _hyperscript does not shy away from using punctuation when appropriate. We'll come across quite a lot of new syntax we use as we go. To get our feet wet, here's an annotated version of the script above:

```
on click -- Event listener
  increment -- This command (built into the language) increments things
    the -- "the" is ignored
    textContent of -- "b of a" and "a's b" are alternative forms of "a.b"
    the previous -- "previous x" == element before me in the DOM that matches x
    <output /> -- A CSS selector is wrapped between "<" and "/>"
```

The `previous` keyword (and the accompanying `next`) are an example of how _hyperscript makes DOM operations easier. As an exercise, you can try to implement a function `previous(selector: string): Node` that does the same.

9.4.1. _hyperscript in action: a keyboard shortcut

Since our keyboard shortcut focuses a search input, let's put the code on that search input. Here it is:

```
<input id="search" name="q" type="search" placeholder="Search Contacts">
```

We begin with an event listener, which, as we explained, starts with `on`:

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
  _="on keydown [shiftKey and code is 'KeyS'] ❶❷❸❹
       -- ...">
```

❶ The square bracket notation is *event filtering* --- any event for which the expression inside the brackets is falsey will be ignored by this listener.

❷ Inside the event filter, properties of the event can be directly accessed.

❸ `and` is `&&` in JavaScript.

❹ `is` is `==` in JavaScript.

We are using event filtering to listen to only the events we are interested in, i.e. the user pressing kbd:[Shift+S]. There is a problem, however: Keyboard events will only be sent to this input element if it is already focused. We need to attach the listener to the whole window instead. No problem:

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
  _="on keydown [shiftKey and code is 'KeyS'] from the window ❶
       -- ...">
```

❶ "from" is part of the "on" feature and lets us listen to events from other objects.

We can attach the listener to the body while keeping its code on the element it logically relates to. Let's actually focus that element now:

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
  _="on keydown[shiftKey and code is 'KeyS'] from the window
       focus() me"> ❶❷
```

❶ Any method of any object can be used as a command. (This is called a "pseudocommand" in _hyperscript lingo). This line is equivalent to `me.focus()` (which is also valid syntax in _hyperscript).

❷ "me" refers to the element that the script is written on.

There's our code! Surprisingly terse for an English-like programming language, compared to the equivalent JavaScript:

```
const search = document.querySelector("#search")
window.addEventListener("keydown", e => {
  if (e.shiftKey && e.code === "KeyS") search.focus();
})
```

9.4.2. Why a new programming language?

Being an interpreter written in JavaScript, the _hyperscript runtime has a lot of overhead. One might wonder why it isn't implemented as a JavaScript library. A new programming language allows us to provide features and fix warts in a way that wouldn't be possible otherwise:

Async transparency

In _hyperscript, asynchronous functions (i.e. functions that return `Promise` instances) can be invoked as if they were synchronous. Changing a function from sync to async does not break any _hyperscript code that calls it. This is achieved by checking for a Promise when evaluating any expression, and suspending the running script if one exists

(only the current event handler is suspended and the main thread is not blocked). JavaScript does not allow us to hook into expression evaluation at the level of granularity needed to achieve this.

Array property access

In _hyperscript, accessing a property on an array (other than `length` or a number) will return an array of the values of property on each member of that array --- in other terms, `a.name` is equivalent to `a.map(el ⍰ el.name)`. jQuery has a similar feature, but only for its own data structure.

9.4.3. Reusable behavior in _hyperscript

The main mechanism for reuse in _hyperscript is _behaviors_ --- named collections of *features* (event listeners, function definitions etc.) that can be *installed* as follows:

```
<div _="install ToggleableMenu(button: .menu-button in me, menu: #contents)"> ❶
  <button class="menu-button">Options</button>
  <div id="contents">
```

❶ Behaviors can accept arguments.

A nice aspect of _hyperscript behaviors is that any element's script can be refactored into a reusable behavior on a copy-paste basis:

Listing 6. 106. The search bar keyboard shortcut code, extracted into a behavior

```
behavior SearchShortcut
  on keydown[shiftKey and code is 'KeyS'] from the window
    focus() me
  end
end
```

Prime examples of behavior usage can be found on Ben Pate's *Hyperscript Widgets* collection (https://github.com/benpate/hyperscript-widgets). Reproduced here with minor cleanup is a rich text editor implemented in 68 lines:

Listing 6. 107. wysiwyg._hs

```
behavior wysiwyg(name)

    -- WYSIWYG setup
```

```
init
    -- save links to important DOM nodes
    set :form to closest <form />
    set :input to form.elements[name]
    set :editor to first .wysiwyg-editor in me

    -- configure related DOM nodes
    add [@tabIndex=0] to :editor
    add [@contentEditable=true] to :editor

    tell <button/> in me
        add [@type="button"]
    end

-- Clicking a toolbar button triggers a command on the content
on click(target)
    if target's @data-command is null then
        set target to the closest <[data-command]/> to target
        if target is null then
            exit
        end
    end

    set command to target's @data-command

    -- special handling for inertLink
    if command is "createLink" then
        get prompt("Enter Link URL")
        call document.execCommand(command, false, result)
        exit
    end

    -- fall through to all other commands
    set value to target's @data-command-value
    call document.execCommand(command, false, value)
end

-- Show the toolbar when focused
on focus(target) from the .wysiwyg-editor in me
remove @hidden from the .wysiwyg-toolbar in me
end

-- Hide the toolbar when blured
on blur from the .wysiwyg-editor in me
    wait 200ms
```

```
        if (<:focus/> in me) is empty then
      add [@hidden=true] to the .wysiwyg-toolbar in me
        end
    end

    -- Autosave the WYSIWYG after 15s of inactivity
    on input debounced at 15s
        send updated to form
    end

    -- Autosave the WYSIWYG whenever it loses focus
    on blur from the .wysiwyg-editor in me
        send updated to form
    end

    -- Push the value directly into the XHR request before it's sent.
    on htmx:configRequest(parameters) from closest <form/>
        set value to the editor's innerHTML
        Object.defineProperty(parameters, name, {value: value, writable:'true'})
    end
```

You can try the editor on https://benpate.github.io/hyperscript-widgets/wysiwyg/.

_hyperscript, being a whole programming language, goes a lot deeper than what was introduced here. Further information is available at https://hyperscript.org/docs.

> In keeping with general htmx principles, we will endeavor to create code that is:
>
> - Usable
> - Accessible
> - Un-Scalable
>
> — https://benpate.github.io/hyperscript-widgets/

9.5. Using off-the-shelf components

9.5.1. Off-the-shelf components in action: drag to reorder

9.6. Events and the DOM

One thing that you will notice in all the scripting that we add to Contact.app is the heavy use of *events*. This is not an accident; scripting in a Hypermedia Driven Application should

---

be oriented around events --- mostly listening to DOM events, but also dispatching custom events. Since htmx allows requests to be triggered upon any type of event, custom events provide an excellent bridge between client-side scripts and the hypermedia exchanges that define a RESTful Hypermedia Driven Application.

Another thing you might notice about the scripting examples is that they all mutate the DOM in some way, such as showing or hiding elements, changing the text content of an element, or moving focus. In many cases this change in state isn't synchronized with the server, which may, at first, seem to discredit our aim of using hypermedia is the engine of application state.

Both the use of events and the prevalence of DOM mutations point to the fact that the purpose of scripting in a HDA is to enhance UI interaction. Use of events reflects that we are mainly concerned with responding to the user's actions. DOM mutations make up a large portion of our code because we are concerned with UI as opposed to business logic when we write scripts. The state retained by client-side scripts should be an extension of state retained by the browser (e.g.: the value of an input element before it is submitted), ephemeral, not closely tied to the application domain, and *ephemeral*. Scripts may use tools like localStorage to keep some user-specific data, what they should not do is alter a canonical data store without going through a hypermedia channel. (As a **very rough** rule of thumb, this means that scripts should avoid making non-GET requests to your server).

9.7. Being pragmatic

> In case of conflict, consider users over authors over implementors over specifiers over theoretical purity.
>
> — W3C, HTML Design Principles § 3.2 Priority of Constituencies

The sad truth is that there will never be a general theory of web development. Any guideline, methodology, or rule of thumb will hit degenerate cases. When this happens, there are a few ways to react:

Denial

Why would we want to implement this feature, anyway? Invent reasons why the problem should not be solved.

Anger

Vehemently refuse to abandon your principles and implement the feature with your method, without regard for the consequences.

Bargaining

Try to invent a new theory to accommodate the feature. It will be incoherent.

Depression

Fantasize about leaving the software industry.

Acceptance

Implement the feature the way you always knew it should be. Leave a comment for any future developer who might be compulsed to "refactor" it.

9.8. Summary

Use progressive enhancement.

Maximize locality of behavior, sometimes at the expense of separation of concerns. Remember that "concerns" are not the same thing as filetypes.

If you’re mostly going to write reusable, generalized components: use vanilla JavaScript with Alpine.js.

If you’re mostly going to write one-off, specialized components: use Alpine.js or _hyperscript.

If you need a common UI pattern that isn’t built into HTML: use a library. If you’re going to write such a library yourself, use vanilla JS with RSJS.

Alpine lets things auto-update based on changes to state and lets you use the programming language known by the most people.

_hyperscript offers a concise, readable syntax, especially for DOM operations, and makes async operations easy.

Events are cool.

Do not use scripts to directly modify system state. Reserve it for UI state. = Hypermedia In

Action :chapter: 9 :sectnums: :figure-caption: Figure 6. :listing-caption: Listing 6. :table-caption: Table 6. :sectnumoffset: 8 :leveloffset: 1 :sourcedir: ../code/src :source-language:

10. Data APIs & Hypermedia Driven Applications

This chapter covers

- Data APIs and how the contrast with hypermedia APIs
- Adding a JSON-Based Data API to our application
- Adding hypermedia controls to our JSON Data API

Data APIs

In this book we have been focusing on using hypermedia to build Hypermedia Driven Applications. In doing so we are following the original networking architecture of the web, building a RESTful system.

However, today, many web applications are not built using this approach. Instead, they use a front end library such as React to interact with JSON API on the server. This JSON API typically does not use hypermedia, but, rather is a *Data API*, that is, it simply returns structured domain data to the client, for the client itself to interpret.

Unfortunately, today, for historical reasons, these JSON APIs are often referred to REST APIs, despite the fact that they are, using the original definition of that term, not actually RESTful.

> 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

The story of how REST came to mean "JSON APIs" in the industry is a long and sordid long one, and beyond the scope of this book. However, if you are interested, you can refer

to an essay entitled "How Did REST Come To Mean The Opposite of REST?" on the htmx website.

In this book we will use the term "Data API" to describe these JSON APIs, while acknowledging that the industry will likely continue to call them "REST APIs" for the foreseeable future.

Now, believe it or not, we *have* been creating a RESTful API for our web application. This may sound confusing to you. API? We have just been creating a web application, with paths that return HTML. How is that a RESTful API?

It turns out that our web application is, indeed, providing an API. It just happens to be an API that a *hypermedia client*, that is, a browser, understands. We are building an API for the browser to interact with over HTTP, and, thanks in no small part to HTML, the hypermedia we are using, we are building a RESTful API. Building web applications like this is so natural and simple that you might not think of it as an API at all, but I assure you, it is!

10.1. Hypermedia APIs & Data APIs

So, we have a hypermedia API for Contact.app. Should we include a Data API as well?

Sure! The existence of a hypermedia API *in no way means* that you can't also have a Data API. In fact, this is a common situation in traditional web applications: there is a "web app" that is entered through one URL, say `https://mywebapp.example.com/`, and then a separate JSON API that is accessible through another URL, say `https://api.mywebapp.example.com/v1`.

This is a perfectly reasonable way to split up the hypermedia interface to your application and the Data API you provide to other, non-hypermedia clients.

Now, why would you want to include a Data API along with a hypermedia API? Well, because non-hypermedia clients want to interact with your application as well, of course! For example, perhaps you have a mobile application that isn't built using HyperView (it's ok, we forgive you). Or maybe you have automated script that needs to interact with the system on a regular basis. Or perhaps there are 3rd party clients who wish to integrate with your system's data in some way.

For all of these use cases, a JSON Data API makes sense: these are not hypermedia clients so presenting them with a hypermedia API like we have would be inefficient and complicated. A simple JSON Data API fits the bill for what we want and, as always, we recommend using the right tool for the job.

"You Want Me To Parse HTML!?!"

A confusion we often run into in online discussions is that, when we advocate a hypermedia approach to building web applications, people think we mean that they should parse the HTML responses from the server to dump the data into their SPA framework or mobile applications.

This is, of course, silly. What we mean, instead, is that you should consider using a hypermedia API *with a hypermedia client*, like the browser, interpreting the hypermedia response itself and presenting it to the user. If you are writing code to tease apart hypermedia, you are probably doing it wrong.

A lot of confusion around this comes, again, from not understanding that an HTML response from a server *is* and API response, just a very different one than most people think of when writing software.

10.1.1. Differences Between Hypermedia APIs & Data APIs

So, OK, we *are* going to have a Data API for our application. At this point, some developers may be wondering: why have both? Why not have a single API, the JSON Data API, and have multiple clients use this one API to communicate with it? Isn't it redundant to have both types of APIs for our application?

It's a reasonable point: we do advocate having multiple APIs to your web application if necessary and, yes, this may lead to some redundancy in code. However, there are distinct advantages to both sorts of API and, even more so, distinct requirements for both sorts of APIs. By supporting both of these types of APIs separately you can get the strengths of both, while keeping their varying styles of code and infrastructure needs cleanly split out.

Let's consider some of the needs of a JSON Data API:

- It must remain stable over time: you cannot change the API willy nilly or you risk breaking clients that use the API and expect certain end points to behave in certain ways

- It must be versioned: related to the first point, when you do make a major change, you need to version the API so that clients that are using the old API continues to work
- It should be rate limited: since data APIs are often used by other clients, not just your own internal web application, requests should be rate limited, often by user, in order to avoid a single client overloading the system
- It should be a general API: since the API is for *all* clients, not just for your web application, you should avoid specialized end points that are driven by your own application needs. Instead, the API should be general and expressive enough to satisfy as many potential client needs as possible.
- Authentication for these sorts of API is typically token based, which we will discuss in more detail later

Contrast these needs with that of a hypermedia API:

- There is no need to remain stable over time: all URLs are discovered via HTML responses, so you can be much more aggressive in changing the shape of a hypermedia API
- This means that versioning is not an issue, another strength of the hypermedia approach
- Rate limiting probably isn't as important beyond the prevention of Distributed Denial of Service (DDoS) attacks
- The API can be *very specific* to your application needs: since it is designed only for your particular web application, and since the API is discovered through hypermedia, you can add and remove highly tuned end points for specific features or optimization needs in your application
- Authentication is typically managed through a session cookie established by a login page

These two different types of APIs have different strengths and needs, so it makes sense to use both. The hypermedia approach can be used for your web application, allowing you to specialize the API for the "shape" of your application. The Data API approach can be used for other, non-hypermedia clients like mobile, integration partners, etc.

Note in particular that, by splitting these two APIs apart from one another, you reduce the pressure that running your web application through your general Data API produces to be

constantly changing the API to address application needs. Rather than being thrashed around with every feature change, your Data API can focus on remaining stable and reliable. This is the core strength of this split API approach, in our opinion.

10.2. Adding a JSON Data API To Contact.app

Alright, so how are we going to add a JSON Data API to our application? One approach, popularized by the Ruby on Rails web framework, is to use the same URL endpoints as your hypermedia application, but use the HTTP `Accept` header to determine if the client wants a JSON representation or an HTML representation. The HTTP `Accept` header allows a client to specify what sort of Multipurpose Internet Mail Extensions (MIME) types, that is file types, it wants back from the server: JSON, HTML, text and so on.

So, if the client wanted a JSON representation of all contacts, they might issue a `GET` request that looks like this:

Listing 6. 108. A Request for a JSON Representation of All Contacts

```
Accept: application/json

GET /contacts
```

If we adopted this pattern then our request handler for `/contacts/ would need to be updated to inspect this header and, depending on the value, return a JSON rather than HTML representation for the contacts. Ruby on Rails has support for this pattern baked into the language, making it very easy to switch on the requested MIME type.

Unfortunately, our experience with this pattern has not been great, for reasons that should be clear given the differences we outlined between Data and hypermedia APIs: they have different needs and often take on very different "shapes", and trying to pound them into the same set of URLs ends up creating a lot of tension in the application code.

So, here, we advocate for applying the Separation of Concerns software design principle and breaking the JSON Data API out to its own set of URLs. This will allow us to evolve the two APIs separately from one another, and give us room to improve each independently in a manner consistent with their own individual strengths.

10.2.1. Picking a Root URL For Our API

Given that we are going to split our JSON Data API routes out from our regular hypermedia routes, where should we place them? One important consideration here is that we want to make sure that we can version our API cleanly in some way, regardless of the pattern we choose. Looking around, a lot of places end up using a sub-domain for their apis, something like `https://api.mywebapp.example.com` and, in fact, often encode versioning in the subdomain: `https://vi.api.mywebapp.example.com`.

While this makes sense for large companies, it seems like a bit of overkill for our modest little Contact.app. Rather than using sub-domains, which are a pain for local development, we will use sub-paths within the existing application:

- We will use `/api` as the root for our Data API functionality
- We will use `/api/v1` as the entry point for version 1 of our Data API

If and when we decide to bump the API version, we can move to `/api/v2` and so on.

This approach isn't perfect, of course, but it will work for our simple application and can be adapted to a subdomain approach or various other methods at a later point, when our Contact.app has taken over the internet and we can afford a large team of API developers. :)

10.2.2. Our First JSON Endpoint: Listing All Contacts

Let's add our first Data API End point. It will handle an HTTP `GET` request to `/api/v1/contacts`, and return a JSON list of all contacts in the system. In some ways it will look quite a bit like our initial code for the hypermedia route `/contacts`: we will load all the contacts from the contacts database and then render some text as a response.

We are going to take advantage of a nice feature of Flask: if you simply return an object from a handler, it will serialized (that is, convert) that object into a JSON response. This makes it very easy to build simple JSON APIs in flask!

Here is our code:

Listing 6. 109. A JSON Data API To Return All Contacts

```
@app.route("/api/v1/contacts", methods=["GET"]) ❶
def json_contacts():
    contacts_set = Contact.all() ❷
    contacts_dicts = [c.__dict__ for c in contacts_set] ❸
    return {"contacts": contacts_dicts} ❹
```

❶ We put our JSON Data API in its own path

❷ We aren't going to support paging or filtering, so we can just load all the contacts here

❸ We convert the contacts array into an array of simple dictionary (map) objects, so they can be serialized to JSON easily

❹ We return a simple dictionary that contains the `contacts` property, pointing to this new array. Flask will automatically serialize this dictionary to JSON for us

The second to last line might look a little funky if you are not a python developer, it is called a "list comprehension", but it's just a way to convert or map a list of values, in this case contacts, to a list of dictionaries or maps. Don't worry about the details, we just want you to understand the general idea: load up all the contacts, do some conversions to make them JSON serializeable, and then return that data structure.

With this in place, if we make an HTTP `GET` request to `/api/v1/contacts`, we will see a response that looks something like this:

Listing 6. 110. Some Sample Data From Our API

```
{
  "contacts": [
    {
      "email": "carson@example.comz",
      "errors": {},
      "first": "Carson",
      "id": 2,
      "last": "Gross",
      "phone": "123-456-7890"
    },
    {
      "email": "joe@example2.com",
      "errors": {},
      "first": "",
      "id": 3,
      "last": "",
      "phone": ""
    },
    ...
```

So, you can see, a relatively simple JSON representation of our contacts. Not perfect, but good enough for the purposes of this book. This is certainly good enough to, for example, write an automated script against, if, for example, you wanted to move your contacts to another system on a nightly basis.

10.2.3. Adding Contacts

Let’s move on the next piece of functionality: adding a new contact to the system. Once again, our code is going to look similar in some ways to the code that we wrote for our normal web application. However, here we are also going to see the JSON API and the hypermedia API for our web application begin to obviously diverge.

In the web application, we needed a separate path, `/contacts/new` to host the HTML form for creating a new contact. In the web application we made the decision to issue a `POST` to that same path to keep things consistent.

In the case of the JSON API, there is no such path needed: the JSON API "just is": it doesn’t need to provide any hypermedia representation for creating a new contact. You simply know where to issue a `POST` to to create a contact, likely through some provided

documentation about the API, and that's it.

Because of that fact, we can put the "create" handler on the same path as the "list" handler: `/api/v1/contacts`, but have it respond only to HTTP `POST` requests.

The code here is relatively straight forward: populate a new contact with the information from the `POST` request, attempt to save it and, if it is not successful, show some error messages. Here is the code:

Listing 6. 111. Adding Contacts With Our JSON API

```
@app.route("/api/v1/contacts", methods=["POST"]) ❶
def json_contacts_new():
    c = Contact(None, request.form.get('first_name'), request.form.get('last_name'),
request.form.get('phone'),
                request.form.get('email')) ❷
    if c.save(): ❸
        return c.__dict__
    else:
        return {"errors": c.errors}, 400 ❹
```

❶ This handler is on the same path as the first one for our JSON API, but handles `POST` requests

❷ We create a new Contact based on values submitted with the request

❸ We attempt to save the contact and, if successful, render it as a JSON object

❹ If the save is not successful, we render an object showing the errors, with a response code of `400 (Bad Request)`

In some ways similar to our `contacts_new()` handler from our web application (we are creating the contact and attempting to save it) but in other ways very different:

- There is no redirection happening here on a successful creation, because we are not dealing with a hypermedia client like the browser
- In the case of a bad request, we simply return an error response code, `400 (Bad Request)`. This is in contrast with the web application, where we simply re-render the form with error messages in it.

It is these sorts of differences that, over time, build up and make the idea of keeping your

JSON and hypermedia APIs on the same set of URLs less and less appealing.

10.2.4. Viewing Contact Details

Next let's make it possible for a JSON API client to download the details for a single client. We will naturally use an HTTP `GET` for this functionality and we will follow the convention we established for our regular web application, and put the path at `/api/v1/contacts/<contact id>`, so, for example, if you want to see the details of the contact with the id `42`, you would issue an HTTP `GET` to `/api/v1/contacts/42`.

This code is quite simple:

Listing 6. 112. Getting the Details of a Contact in JSON

```
@app.route("/api/v1/contacts/<contact_id>", methods=["GET"]) ❶
def json_contacts_view(contact_id=0):
    contact = Contact.find(contact_id) ❷
    return contact.__dict__ ❸
```

❶ Add a new `GET` route at the path we want to use for viewing contact details

❷ Look the contact up via the id passed in through the path

❸ Convert the contact to a dictionary, so it can be rendered as JSON response

Nothing too complicated: we look the contact up by ID, provided in the path to the controller, and look that contact up. We then render it as JSON. You have to appreciate the simplicity of this code!

Next, let's add updating and deleting a contact as well.

10.2.5. Updating & Deleting Contacts

As with the create contact API end point, because there is no HTML UI to produce for them, we can reuse the `/api/v1/contacts/<contact id>` path. We will use the `PUT` HTTP action for updating a contact and the `DELETE` action for deleting one.

Our update code is going to look nearly identical to the create handler, except that, rather than creating a new contact, we will look up the contact by ID and update its fields. In this sense we are just combining the code of the create handler and the detail view handler.

Listing 6. 113. Updating A Contact With Our JSON API

```
@app.route("/api/v1/contacts/<contact_id>", methods=["PUT"]) ❶
def json_contacts_edit(contact_id):
    c = Contact.find(contact_id) ❷
    c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email']) ❸
    if c.save(): ❹
        return c.__dict__
    else:
        return {"errors": c.errors}, 400
```

❶ We handle `PUT` requests to the URL for a given contact

❷ Look the contact up via the id passed in through the path

❸ We update the contact's data from the values included in the request

❹ From here on the logic is identical to the `json_contacts_create()` handler

Once again, very regular and, thanks to the built-in functionality in Flask, simple to implement.

Let's look at deleting a contact now. This turns out to be even simpler: as with the update handler we are going to look up the contact by id, and then, well, delete it. At that point we can return a simple JSON object indicating success.

Listing 6. 114. Deleting A Contact With Our JSON API

```
@app.route("/api/v1/contacts/<contact_id>", methods=["DELETE"]) ❶
def json_contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete() ❷
    return jsonify({"success": True}) ❸
```

❶ We handle `DELETE` requests to the URL for a given contact

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

❸ Return a simple JSON object indicating that the contact was successfully deleted

And, with that, we have our simple little JSON Data API to live alongside our regular web application, nicely separated out from the main web application, so it can evolve separately as needed.

10.2.6. Differences Between Our Hypermedia And JSON APIs

Now, we obviously have a lot more to do if we want to make this a production ready JSON API:

- We don't have any rate limiting, which is important for any publicly facing Data API to avoid abusive clients.
- Even more crucially, there is currently no authentication mechanism. (We don't have one for our web application either!)
- We currently don't support paging of our contact data.
- Lots of small issues that we aren't addressing, such as rendering a proper `404 (Not Found)` response if someone makes a request with a contact id that doesn't exist.

A full discussion around all of these topics is beyond the scope of this book, but I'd like to focus in on one in particular, authentication, in order to show the difference between our hypermedia and JSON API. In order to secure our application we need to add authentication, some mechanism for determining who a request is coming from, and also authorization, determining if they have the right to perform the request.

We will set authorization aside for now and consider only authentication.

Authentication in Web Applications

In the HTML web application world, authentication has traditionally been done via a login page that asks a user for their username (often their email) and a password. This password is then checked against a database of (hashed) passwords to establish that the user is who they say they are. If the password is correct, then a *session cookie* is established, indicating who the user is. This cookie is then sent with every request that the user makes to the web application, allowing the application to know which user is making a given request.

HTTP Cookies

HTTP Cookies are kind of a strange feature of HTTP. In some ways they violate the goal of remaining stateless, a major component of the REST-ful architecture: a server will often use a session cookie as an index into state kept on the server "on the side", such as a cache of the last action performed by the user.

Nonetheless, cookies have proven extremely useful and so people tend not to complain about this aspect of them too much (I'm not sure what our other options would be here!) An interesting example of pragmatism gone (relatively) right in web development.

In comparison with the typical web application approach to authentication, a JSON API will typically use some sort of *token based* authentication: an authentication token will be established via a mechanism like OAuth, and that authentication token will then be passed, often as an HTTP Header, with every request that a client makes.

At a high level this is similar to what happens in normal web application authentication: a token is established somehow and then then token is part of every request. However, in practice, the mechanics tend to be wildly different:

- Cookies are part of the HTTP specification and can be easily *set* by an HTTP Server
- JSON Authentication tokens, in contrast, often require elaborate exchange mechanics like OAuth to be established

These differing mechanics for establishing authentication are yet another good reason for splitting our JSON and hypermedia APIs up.

The "Shape" of Our Two APIs

When we were building out our API, we noted that in many cases the JSON API didn't require as many end points as our hypermedia API did: we didn't need a `/contacts/new` handler, for example, to provide a hypermedia representation for creating contacts.

Another aspect of our hypermedia API to consider was the performance improvement we made: we pulled the total contact count out to a separate end point and implemented the "Lazy Load" pattern, to improve the perceived performance of our application.

Now, if we had both our hypermedia and JSON API sharing the same paths, would we want to publish this API as a JSON end point as well?

Maybe, but maybe not. This was a pretty specific need for our web application, and, absent a request from a user of our JSON API, it doesn't make sense to include it for JSON consumers.

And what if, by some miracle, the performance issues with `Contact.count()` that we were addressing with the Lazy Load pattern goes away? Well, in our Hypermedia Drive Application we can simply revert to the old code and include the count directly in the request to `/contacts`. We can remove the `contacts/count` end point and all the logic associated with it. By the miracle of hypermedia, the system will continue to work just fine!

But what if we had tied our JSON API and hypermedia API together, and published `/contacts/count` as a supported end point for our JSON API? In that case we couldn't simply remove the end point: a (non-hypermedia) client might be relying on it!

Once again you can see the flexibility of the hypermedia approach and why separating your JSON API out from your hypermedia API lets you take maximum advantage of that flexibility.

10.2.7. The Model View Controller (MVC) Paradigm

One thing you may have noticed about the handlers for our JSON API is that they are relatively simple and regular. Most of the hard work of updating data and so forth is done within the contact model itself: the handlers act as simple connectors that provide a go-between the HTTP requests and the model.

This is the ideal controller of the Model-View-Controller (MVC) paradigm that was so popular in the early web: a controller should be "thin", with the model containing the majority of the logic in the system.

The Model View Controller Pattern

The Model View Controller design pattern is a classic architectural pattern in software development, and was a major influence in early web development. It is no longer emphasized as heavily, as web development has split into front-end and back-end camps, but most web developers are still familiar with the idea.

Traditionally, the MVC pattern mapped into web development like so:

- Model - A collection of "domain" classes that implement all the logic and rules for the particular domain your application is designed for. The model typically provides "resources" that are then presented to clients as HTML "representations".
- View - Typically views would be some sort of client-side templating system, and would render the aforementioned HTML representation for a given Model instance.
- Controller - The controllers job is to take HTTP requests, convert them into sensible requests to the Model and forward those requests on to the appropriate Model objects. It then passes the HTML representation back to the client as an HTTP response.

Thin controllers make it easy to split your JSON and hypermedia APIs out, because all the important logic lives in the domain model that is shared by both. This allows you to evolve both separately, while still keeping logic in sync with one another. With properly built "thin" controllers and "fat" models, keeping two separate APIs both in sync and yet still evolving separately is not as difficult or as crazy as it might sound at first.

10.3. Summary

- In this chapter we saw that a Hypermedia Driven Application can have a JSON Data API as well
- Hypermedia APIs and JSON Data APIs have different needs and shapes
- By properly factoring your back end code, much of the logic can be shared between the two APIs
- By splitting your APIs into both a Hypermedia API and a JSON Data API, you can evolve both without interferring with one another, allowing you to, for example, change

your Hypermedia API dramatically without breaking your JSON API clients = Hypermedia In Action :chapter: 10 :sectnums: :figure-caption: Figure 6. :listing-caption: Listing 6. :table-caption: Table 6. :sectnumoffset: 9 :leveloffset: 1 :sourcedir: ../code/src :source-language:

11. Creating A Dynamic Archive UI

This chapter covers

- Creating a dynamically updated download UI using hypermedia
- Adding smooth animations to a progress bar
- Triggering a file download with a response header

A Dynamic Archive UI

We've come a long way from our plain old traditional web application at this point: we've added active search, bulk delete (with some nice animations) and a slew of other features, to say nothing of the hyperview-based mobile application we have built. I hope you'll agree that we have reached a level of interactivity that most people would assume requires some sort JavaScript framework, but we've done nearly all of it with good old hypermedia and a bit of scripting on the side.

However, despite the wonderful reception of Contact.app in the world, one feature keeps coming up again and again: users would like to be able to download all their contacts, preferably in an easy-to-use JSON format.

This is a reasonable request and another team has been working on the back-end support for doing exactly this. There is one problem though: the archive takes a bit of time to prepare and export, typically on the order of five to 10 seconds, but sometimes longer.

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

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

Just blocking and waiting for the action to complete is certainly the easy way to handle it,

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

The second option, starting the action in a separate, asynchronous manner (say, by starting a thread, or submitting it to a job runner system) is much nicer: you can respond immediately and the user doesn't need to sit there wondering what's going on. But the question is, what do you respond with?

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

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

All of these work, but they sure aren't great user experiences, are they?

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

Now, at this point, surely we are beyond what can be achieved using only hypermedia, right? Well, we wouldn't have a whole chapter on this topic if that were the case, would we? We'll need to push htmx pretty hard to make this all work, but when it is done it won't be *that* much code, and it will give us the user interface that we want.

UI Requirements

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

As I mentioned earlier, thankfully another team has been working on the actual archive

process, and they have given us a class that we can work with, `Archiver`, that implements all the functionality that we need. In particular, it gives us the following methods:

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

Not a terribly complicated API, the only somewhat tricky aspect to it is that the `run()` method is non-blocking: it starts a background job to do the actual archiving and returns immediately.

Beginning Our Implementation

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

So, in getting down to building the UI, the first thing I want to note is that the UI is largely self-contained: we want to replace the button with the download progress bar, and then the link to download the results of the archive process. Everything will all be in one place in the UI, which is a strong hint that we want to create a new template to handle this little sub-section of the application. Let’s call this template `archive_ui.html`.

Another thing that jumps out at me is that we are going to want to replace the entire download UI in multiple cases. Since we want to do that, it makes sense to wrap the entire UI in a `div` tag, and then use that `div` as the target for all our operations. So let’s get our new template going with the following content:

Listing 6. 115. Our Initial Archive UI Template

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

</div>
```

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

❷ Replace the entire div every time using `outerHTML`

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

Listing 6. 116. Adding The Button

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

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

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

Listing 6. 117. Our Initial Archive UI Template

```
{% block content %}

    {% include 'archive_ui.html' %} ❶

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

❶ This template will now be included in the main template

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

Adding the POST End Point

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

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

This is a very common pattern: we put all the different UIs for a given conceptual "chuck" of the user interface into a single template, and conditionally render the appropriate interface. This keeps everything together and makes it very easy to understand how the UIs interact with one another.

Since we are going to conditionally render different user interfaces based on the state of the archiver, we will need to pass the archiver out to the template. So, again: we need to invoke `run()` and then pass the archiver out to the template for conditional rendering. Here is what the code looks like:

Listing 6. 118. Server Side Code To Start The Archive Process

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

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

❷ Look up the Archiver

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

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

Conditionally Rendering A Progress UI

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

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

Listing 6. 119. Adding Conditional Rendering

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

</div>
```

❶ Only render button if the status is "Waiting"

❷ Render different content when status is "Running"

❸ For now, just some text saying things are Running

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

Ouch:

Listing 6. 120. Something Went Wrong

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

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

Listing 6. 121. Including The Archiver When We Render index.html

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

❶ Pass through archiver to the main template

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

11.1. Polling

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

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

The technique we want to use here is called "polling", where we issue a request on an interval and update the UI based on the new state of the server. Polling has a bit of a bad rap, and it isn't the sexiest technique in the world: today developers might look at a more advanced technique like WebSockets or Server Sent Events (SSE) to address this situation. But, say what one will, polling *works* and it is drop-dead simple. You need to be careful to make sure you don't overwhelm you system with polling requests, but, with a bit of care, you can create a reliable, passively updated component in your UI.

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

example:

Listing 6. 122. Fixed Interval Polling

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

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

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

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

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

11.1.1. Using Polling To Update The Archive UI

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

Listing 6. 123. A CSS-based Progress Bar

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

❶ The width of the inner element corresponds to the progress

This CSS-based progress bar has two components: an outer `div` that provides the wire

frame for the progress bar, and and inner `div` that is the actual progress bar indicator. We set the width of the inner progress bar to some percentage (note we need to multiply the `progress()` result by 100 to get a percentage) and that will make the progress indicator the appropriate width within the parent div.

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

Listing 6. 124. The CSS For Our Progress Bar

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

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

Which ends up rendering like this:

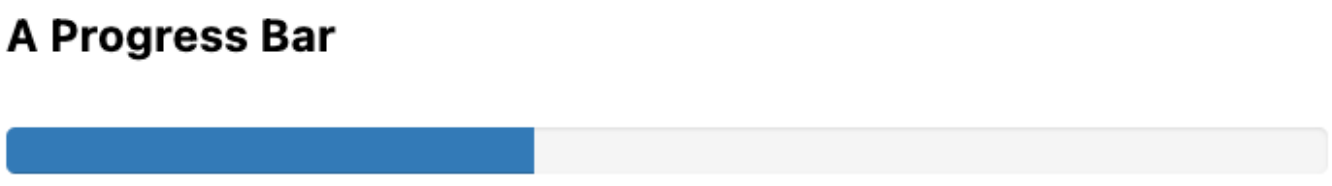


Figure 6. 9. Our CSS-Based Progress Bar

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

Listing 6. 125. Adding The Progress Bar

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

❶ Our shiny new progress bar

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

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

Listing 6. 126. Implementing Load Polling

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

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

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

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

Listing 6. 127. Handling Progress Updates

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

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

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

Simple, like so much else with hypermedia!

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

11.1.2. Downloading The Result

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

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

Listing 6. 128. Rendering A Download Link When Archiving Completes

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

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

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

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

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

The final step is to handle the `GET` request to `/contacts/archive/file`. We want to send the file that the archiver created down to the client. We are in luck: flask has a very simple mechanism for sending a file as a downloaded response: the `send_file()` method. We can pass this method the path to the archive file that the archiver created, the name of the file that we want the browser to create, and if we want it sent "as an attachment", which will set the appropriate HTTP response headers to trigger the browsers downloading behavior.

Listing 6. 129. Sending A File To The Client

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

❶ send the file to the client

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

11.2. Smoothing Things Out: More On The htmx Swap Model

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

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

If you look back at our CSS definition of the `.progress-bar` class, you will see the following transition definition in it: `transition: width .6s ease;`. This means that

when the width of the progress bar is changed from, say 20% to 30%, the browser will animate over a period of .6 seconds using the "ease" function (which has a nice accelerate/decelerate effect).

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

11.2.1. Settling

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

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

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

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

Listing 6. 130. Smoothing Things Out

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

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

All we had to do was add a simple `id` attribute and viola, a much smoother user experience!

11.3. Dismissing The Download UI

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

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

Listing 6. 131. Clearing The Download

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

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

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

Here is the server side code:

Listing 6. 132. Resetting The Download

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

❶ Call `reset()` on the archiver

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

11.4. Auto-Download

One pattern that I see sometimes on the web is "auto-downloading" where a file is created and then, when it is ready, the system automatically downloads the file. We can add that functionality quite easily to our application with a bit of hyperscript.

What we want to do is, when the download link renders, automatically click on the link for the user. The hyperscript will read basically just like that:

Listing 6. 133. Auto-Downloading

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

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

Note that the scripting here is simply *enhancing* the existing hypermedia, rather than replacing it with a non-hypermedia request. This is hypermedia-friendly scripting!

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

11.5. Summary

- In this chapter we built a sophisticated user interface to interact with a non-blocking, asynchornous back end process: creating an archive of all contacts in our application
- We saw a few different ways to do polling in htmx, and settled on using "load polling" for our situation
- We saw how the htmx swap mechanism enables CSS transitions when an element has a stable ID in new pieces of content, and we used that to smooth out the progress bar in our application
- We used a bit of hypermedia-friendly scripting to trigger an auto-download when the

archive progress completes

12. Hypermedia In Action

13. Developing With htmx

This chapter covers

- The details of htmx attributes
- Events & htmx
- HTTP Requests & htmx
- Updating Other Content
- Debugging htmx Applications
- Security Considerations
- Configuring htmx

Getting Deeper Into htmx

In this chapter we are going to look more deeply into htmx. We've accomplished quite a bit with what we've learned so far, but, when you are developing Hypermedia Driven Applications, there are likely to be situations that arise that require additional functionality to address cleanly. We will go over some less commonly used attributes in htmx, as well as expand on the details of some attributes we have already used.

Additionally, we will look at the functionality that htmx offers beyond simple HTML attributes: how htmx extends standard HTTP request and responses, how htmx works with (and produces) events, and how to approach situations where there isn't a simple, single target on the page to be updated.

Finally, we will take a look at practical considerations when doing htmx development: how to debug htmx-based applications effectively, security considerations you will need to take into account when working with htmx, and how to configure the behavior of htmx.

13.1. htmx Attributes

Thus far we have, we've used about fifteen different attributes from htmx in our application. The most important ones have been:

Attribute	Use
`hx-get`, `hx-post`, etc.	To specify the AJAX request an element should make
`hx-trigger`	To specify the event that triggers a request
`hx-swap`	To specify how to swap the returned HTML content into the DOM
`hx-target`	To specify where in the DOM to swap the returned HTML content

Let's do a deep dive on two of these attributes, `hx-swap` and `hx-trigger`, because they support a large number of options that might be useful when you are creating more advanced Hypermedia Driven Applications.

13.1.1. `hx-swap`

The `hx-swap` attribute is often not included on elements that issue htmx-driven requests. This is because, for many cases, the default behavior, `innerHTML`, which swaps the inner HTML of the element, is fine. Of course, we have seen cases where we wanted to override this behavior and use `outerHTML`, for example. And, in chapter 3, we discussed some other swap options beyond these two, `beforebegin`, `afterend`, etc.

In chapter 5, we also looked at the `swap` delay modifier for `hx-swap`, which allowed us to fade some content out before it was removed from the DOM.

In addition to these, `hx-swap` also supports the following modifiers:

Modifier	Use
`settle`	Like `swap`, this allows you to apply a specific delay between when the content has been swapped into the DOM and when its attributes are "settled", that is, updated from their old values (if any) to their new values.

Modifier	Use
`show`	Allows you to specify an element that should be shown (that is, scrolled into the viewport of the browser if necessary) when a request is completed
`scroll`	Allows you to specify a scrollable element (that is, an element with scrollbars), that should be scrolled to the top or bottom when a request is completed
`focus-scroll`	Allows you to specify that htmx should scroll to the focused element when a request completes. (This defaults to false)

So, for example, if we had a button that issued a `GET` request, and we wished to scroll to the top of the `body` element when the request had completed, we would write the following HTML:

Listing 11. 134. Scrolling To The Top Of The Page

```
<button hx-get="/contacts" hx-target="#content-div"
        hx-swap="innerHTML show:body:top"> ❶
  Get Contacts
</button>
```

❶ This tells htmx to show the top of the body after the swap occurs

More details and examples can be found online at the documentation page for `hx-swap`: https://htmx.org/attributes/hx-swap/

13.1.2. `hx-trigger`

Like `hx-swap`, `hx-trigger` can often be omitted when you are using htmx, because the default behavior is typically what you want anyway. Recall the default triggering events are determined by an element's type:

- 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

There are times, however, when you want a more elaborate trigger specification. A classic example was the active search example we implemented in Contact.app:

Listing 11. 135. The Active Search Input

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

❶ An elaborate trigger specification

This example took advantage of two modifiers available for the `hx-trigger` attribute:

Modifier	Use
`delay`	Allows you to specify a delay to wait before a request is issued. If the event occurs again, the first event is discarded and the timer resets. This allows you to "debounce" requests.
`changed`	Allows you to specify that a request should only be issued when the `value` property of the given element has changed

`hx-trigger` has quite a few additional modifiers. This makes sense, because events are fairly complex and we want to be able to take advantage of all the power they offer. (We will discuss events in more detail below.)

Here are the other modifiers available on `hx-trigger`:

Modifier	Use
`once`	The given event will only trigger a request once

Modifier	Use
`throttle`	Allows you to throttle events, only issuing them once every certain interval. This is different than `delay` in that the first event will trigger immediately, but any following events will not trigger until the throttle time period has elapsed
`from`	A CSS selector that allows you to pick another element to listen for events on. We will see an example of this used later in the chapter.
`target`	A CSS selector that allows you to filter events to only those that occur directly on a given element. In the DOM, events "bubble" to their parent elements, so a `click` event on a button will also trigger a `click` event on a parent `div`, all the way up to the `body` element. Sometimes you want to specify an event directly on a given element, and this attribute allows you to do that.
`consume`	If this option is set to `true`, the triggering event will be cancelled and not propagate to parent elements.

Modifier	Use
`queue`	This option allows you to specify how events are queued in htmx. By default, when htmx receives a triggering event, it will issue a request and start an event queue. If the request is still in flight when another event is received, it will queue the event and, when the request finishes, trigger a new request. By default, it only keeps the last event it receives, but you can modify that behavior using this option: for example, you can set it to `none` and ignore all triggering events that occur during a request.

Filters

The `hx-trigger` attribute allows you to specify a *filter* to events by using square brackets enclosing a JavaScript expression after the event name.

Let's say you have a complex situation where contacts should only be retrievable in certain situations, and you have a JavaScript function, `contactRetrievalEnabled()` that returns a boolean, `true` if contacts can be retrieved and `false` otherwise. You want to gate a button that issues a request to `/contacts` on this function. To do this using an event filter in htmx, you would write the following HTML:

Listing 11. 136. The Active Search Input

```
<script>
  function contactRetrievalEnabled() {
      ...
  }
</script>
<button hx-get="/contacts" hx-trigger="click[contactRetrievalEnabled()]"> ❶
  Get Contacts
</button>
```

❶ the event filter, calling `contactRetrievalEnabled()`

The button will not issue a request if `contactRetrievalEnabled()` returns false, allowing you to dynamically control when the request will be made. Common situations that call for an event trigger are:

- Only issue a request when a certain element has focus
- Only issue a request when a given form is valid
- Only issue a request when a set of inputs have specific values

Using event filters, you can use whatever logic you'd like to filter requests by htmx.

Synthetic Events

In addition to these modifiers, `hx-trigger` offers a few "synthetic" events, that is events that are not part of the regular DOM API. We have already seen `load` and `revealed` in our lazy loading and infinite scroll examples, but htmx also gives you an `intersect` event that triggers when an element intersects its parent element.

This synthetic event uses the modern Intersection Observer API, which you can read more about here: https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API

Intersection gives you much finer grained control over exactly when a request should be triggered. For example, you can specify a threshold and specify that the request should only be issued when an element is 50% visible.

The `hx-trigger` attribute certainly is the most complex on in htmx, and more details and examples can be found online its documentation page: https://htmx.org/attributes/hx-trigger/

13.1.3. Other Attributes

htmx offers many other less commonly used attributes for fine-tuning the behavior of your Hypermedia Driven Application. Here are some of the most useful ones:

Attribute	Use
hx-push-url	"Pushes" the request URL (or some other value) into the navigation bar

Attribute	Use
hx-preserve	Preserves a bit of the DOM between requests (the original content will be kept, regardless of what is returned)
hx-sync	Synchronized requests between two or more elements
hx-disable	Disables htmx behavior on this element and any children. We will discuss this more below in the security section.

Let's take a look at `hx-sync`, which allows us to synchronize AJAX requests between two or more elements. Consider a simple case where we have two buttons that both target the same element on the screen:

Listing 11. 137. Two Competing Buttons

```
<button hx-get="/contacts" hx-target="body"> ❶
  Get Contacts
</button>
<button hx-get="/settings" hx-target="body"> ❶
  Get Settings
</button>
```

This is fine and will work, but what if a user clicks the "Get Contacts" button and then the request takes a while to respond? And, in the meantime the user clicks the "Get Settings" button? In this case we would have two requests in flight at the same time.

If the `/settings` request finished first and displayed the user's setting information, they might be very surprised if they began making changes and then, suddenly, the `/contacts` request finished and replaced the entire body with the contacts instead!

To deal with this situation, we might consider using an `hx-indicator` to alert the user that something is going on, making it less likely that they click the second button. But if we really want to guarantee that there is only one request at a time issued between these two buttons, the right thing to do is to use the `hx-sync` attribute. Let's enclose both buttons in a `div` and eliminate the redundant `hx-target` specification by hoisting the attribute up to

that `div`. We can then use `hx-sync` on that div to coordinate requests between the two buttons.

Here is our updated code:

Listing 11. 138. Syncing Two Buttons

```
<div hx-target="body"  ❶
     hx-sync="this">  ❷
    <button hx-get="/contacts"> ❶
      Get Contacts
    </button>
    <button hx-get="/settings"> ❶
      Get Settings
    </button>
</div>
```

❶ Hoist the duplicate `hx-target` attributes to the parent `div`

❷ Synchronize on the parent `div`

By placing the `hx-sync` attribute on the `div` with the value `this`, we are saying "Synchronize all htmx requests that occur within this `div` element with one another." This means that if one button already has a request in flight, other buttons within the `div` will not issue requests until that has finished.

The `hx-sync` attribute supports a few different strategies that allow you to, for example, replace an existing request in flight, or queue requests with a particular queuing strategy. You can find complete documentation, as well as examples, at the documentation page for `hx-sync`: https://htmx.org/attributes/hx-sync/

As you can see, htmx offers a lot of attribute-driven functionality for more advanced Hypermedia Driven Applications. A complete reference for all htmx attributes can be found at https://htmx.org/reference/#attributes

13.2. Events

We have been working with events in htmx primary via the `hx-trigger` attribute. This has proven to be a powerful mechanism for driving our application using declarative, HTML-friendly syntax. However, there is more to events and htmx than just `hx-trigger`.

13.2.1. htmx-generated Events

It turns out that, in addition to making it easy to *respond* to events, htmx also *emits* many useful events. You can use these events to add more functionality to your application, either via htmx itself, or by way of scripting.

Here are some of the most commonly used events in htmx:

Event	Description
`htmx:load`	Triggered when new content is loaded into the DOM by htmx
`htmx:configRequest`	Triggered before a request is issued, allowing you to programmatically configure the request (or cancel it entirely)
`htmx:afterRequest`	Triggered after a request has responded
`htmx:abort`	A custom event that can be sent to an htmx-powered element to abort an open request

We have already seen how to use the `htmx:load` event, using the `htmx.onLoad()` API in the sortable.js example, which is probably one of the most common uses of events.

13.2.2. Using The `htmx:configRequest` Event

Let's take a look at how you might use the `htmx:configRequest` event to configure an HTTP request. Consider the following scenario: our server-side team has decided that they want you to include a token for extra validation on every request. The token is going to be stored in `localStorage` in the browser, in the slot `special-token`. The server-side team wants you to include this special token on every request made by htmx, as the `X-SPECIAL-TOKEN` header.

How could you achieve this? One way would be to catch the `htmx:configRequest` event and update the `detail.headers` object with this token from `localStorage`.

In VanillaJS, it would look something like this:

Listing 11. 139. Adding the `X-SPECIAL-TOKEN` Header

```
document.body.addEventListener("htmx:configRequest", function(configEvent){
    configEvent.detail.headers['X-SPECIAL-TOKEN'] = localStorage['special-token']; ❶
})
```

❶ retrieve the value from local storage and set it into a header

As you can see, we add a new value to the `headers` property of the event's detail. After the event handler executes, the `headers` property is read by htmx and used to construct the headers for an AJAX request. So, with this bit of JavaScript code, we have added a new custom header to every AJAX request that htmx makes. Slick!

You can also update the `parameters` property to change the parameters submitted by the request, change the target of the request, and so on. Full documentation for the `htmx:configRequest` event can be found here: https://htmx.org/events/#htmx:configRequest

13.2.3. Canceling a Request using `htmx:abort`

So we can listen for many useful events from htmx, and we can respond to events using `hx-trigger`. What else can we do with events? It turns out that htmx itself listens for one special event, `htmx:abort`. When htmx receives this event on an element that has a request in flight, it will abort the request.

Consider a situation where we have a potentially long-running request to `/contacts`, and we want to offer a way for the users to cancel the request. What we want is a button that issues the request, driven by htmx, of course, and then another button that will send an `htmx:abort` event to the first one.

Here is what the code might look like:

Listing 11. 140. A Button With An Abort

```
<button id="contacts-btn" hx-get="/contacts" hx-target="body"> ❶
  Get Contacts
</button>
<button onclick="document.getElementById('contacts-btn').dispatchEvent(new
Event('htmx:abort'))"> ❷
  Cancel
</button>
```

❶ A normal htmx-driven `GET` request to `/contacts`

❷ JavaScript to look up the button and send it an `htxm:abort` event

So now, if a user clicks on the "Get Contacts" button and the request takes a while, they can click on the "Cancel" button and end the request. Of course, in a more sophisticated user interface, you may want to disable the "Cancel" button unless an HTTP request is in flight, but that would be a pain to implement in pure JavaScript. Thankfully it isn't too bad to implement in hyperscript, so let's take a look at what that would look like:

Listing 11. 141. A hyperscript-Powered Button With An Abort

```
<button id="contacts-btn" hx-get="/contacts" hx-target="body">
  Get Contacts
</button>
<button _="on click send htmx:abort to #contacts-btn
           on htmx:beforeRequest from #contacts-btn remove @disabled from me
           on htmx:afterRequest from #contacts-btn add @disabled to me">
  Cancel
</button>
```

Now we have a "Cancel" button that is disabled only when a request from the `contacts-btn` button is in flight. And we are taking advantage of htmx-generated and handled events, as well as the event-friendly syntax of hyperscript, to make it happen. Not bad!

13.2.4. Server Generated Events

We are going to talk more about the various ways that htmx enhances regular HTTP requests and responses in the next section, but, since it involves events, we are going to discuss one HTTP Response header that htmx supports: `HX-Trigger`. We have discussed before how HTTP requests and responses support *headers*, name-value pairs that contain

metadata about a given request or response. We took advantage of the `HX-Trigger` request header, which includes the id of the element that triggered a given request.

It turns out that there is a *response* header, also named `HX-Trigger` in that htmx supports. This response header allows you to trigger an event on the element that submitted an AJAX request. This turns out to be a powerful way to coordinate elements in the DOM in a decoupled manner.

To see how this might work, lets consider the following situation: we have a button that grabs new contacts from some remote system on the server. We will ignore the details of the server side implementation, but we know that if we issue a `POST` to the `/integrations/1` path, it will trigger a synchronization with the system.

Now, this synchronization may or may not result in new contacts being created. In the case where new contacts *are* created, we want to refresh our contacts table. In the case where no contacts are created, we don't want to refresh the table.

How could we implement this using the `HX-Trigger` response header? Well, we could conditionally add an `HX-Trigger` response header with the value `contacts-updated`, which would trigger the `contacts-updated` event on the button that made the AJAX request to `/integrations/1`. And we can then take advantage of the `from:` modifier of the `hx-trigger` attribute to listen for that event! Now we can effectively trigger htmx request from the server side!

Here is what the client-side code might look like:

Listing 11. 142. The Contacts Table

```
<button hx-post="/integrations/1"> ❶
  Pull Contacts From Integration
</button>

   ...

 <table hx-get="/contacts/table" hx-trigger="contacts-updated from:body"> ❷
   ...
 </table>
```

❶ The response to this request may conditionally trigger the `contacts-updated` event

❷ This table listens for the event and refreshes when it occurs

The table listens for the `contacts-updated` event, and it does so on the `body` element. It listens on the `body` element since the event will bubble up from the button, and this allows us to not couple the button and table together: we can move the button and table around as we like and, via events, the behavior we want will continue to work fine. Additionally, we may want *other* elements or requests to trigger the `contacts-updated` event, so this provides a general mechanism for refreshing the contacts table in our application. Very nice!

Now, we are omitting the server side implementation of this feature in the interest of simplicity, but this gives you and idea of how the `HX-Trigger` response header can be used to coordinate sophisticated interactions in the DOM.

13.3. HTTP Requests & Responses

We have just seen an advanced feature of HTTP responses supported by htmx, the `HX-Trigger` response header, but htmx supports quite a few more headers for both requests and responses. In chapter 5 we discussed the headers present in HTTP Requests. Here some of the more important headers you can use to change htmx behavior with HTTP responses:

Response Header	Description
`HX-Location`	Causes a client-side redirection to a new location
`HX-Push-Url`	Pushes a new URL into the location bar
`HX-Refresh`	Refreshes the current page
`HX-Retarget`	Allows you to specify a new target to swap the response content into on the client side

You can find a reference for all requests and response headers here: https://htmx.org/reference/#headers

13.3.1. HTTP Response Codes

Even more important than response headers, in terms of information conveyed to the client, is the *HTTP Response Code*. We discussed HTTP Response Codes in Chapter 4. By and

large htmx handles various response codes in the manner that you would expect: it swaps content for all 200-level response codes and does nothing for others. There are, however, two "special" 200-level response codes:

- `204 No Content` - When htmx receives this response code, it will *not* swap any content into the DOM (even if the response has a body)
- `286` - When htmx receives this response code to a request that is polling, it will stop the polling

You can override the behavior of htmx with respect to response codes by, you guessed it, responding to an event! The `htmx:beforeSwap` event allows you to change the behavior of htmx with respect to various status codes.

Let's say that, rather than doing nothing when a `404` occurred, you wanted to alert the user that an error had occurred. To do so, you want to invoke a JavaScript method, `showNotFoundError()`. Let's add some code to use the `htmx:beforeSwap` event to make this happen:

Listing 11. 143. Showing a 404 Dialog

```
document.body.addEventListener('htmx:beforeSwap', function(evt) { ❶
    if(evt.detail.xhr.status === 404){ ❷
        showNotFoundError();
    }
});
```

❶ hook into the `htmx:beforeSwap` event

❷ if the response code is a `404`, show the user a dialog

You can also use the `htmx:beforeSwap` event to configure if the response should be swapped into the DOM and what element the response should target. This gives you quite a bit of flexibility in choosing how you want to use HTTP Response codes in your application. Full documentation on the `htmx:beforeSwap` event can be found here: https://htmx.org/events/#htmx:beforeSwap

13.4. Updating Other Content

Above we saw how to use a server-triggered event, via the `HX-Trigger` HTTP response

header, to update a piece of the DOM based on the response to another part of the DOM. This technique addresses the general problem that comes up in Hypermedia Driven Applications: "How do I update other content?" After all, in normal HTTP requests, there is only one "target", the entire screen, and, similarly, in htmx-based requests, there is only one target: either the explicit or implicit target of the element.

If you want to update other content in htmx, you have a few options:

13.4.1. Expanding Your Selection

The first option, and the simplest, is to "expand the target". That is, rather than simply replacing a small part of the screen, expand the target of your htmx-driven request until it is large enough to enclose all the elements that need to updated on a screen. This has the tremendous advantage of being simple and reliable. The downside is that it may not provide the user experience that you want, and it may not play well with a particular server-side template layout. Regardless, I always recommend at least thinking about this approach first.

13.4.2. Out of Band Swaps

A second option, which is a bit more complex, is to take advantage of "Out Of Band" content support in htmx. When htmx receives a response, it will look for top-level content in that response that includes the `hx-swap-oob` attribute on it. That content will be removed from the response, so it will not be swapped into the DOM in the normal manner. Instead, it will be swapped in for the content that it matches, by its id.

Let’s look at an example of this approach. Let’s consider the situation we had above, where a contacts table needs to be updated conditionally, based on if an integration pulls down any new contacts. Previously we solved this by using events and a server-triggered event via the `HX-Trigger` response header.

In this case, instead of using an event, let’s take advantage of the `hx-swap-oob` attribute in the response to the `POST` to `/integrations/1` to "piggy back" the new contacts table content on the response.

Listing 11. 144. The Updated Contacts Table

```
<button hx-post="/integrations/1"> ❶
  Pull Contacts From Integration
</button>

   ...

 <table id="contacts-table"> ❷
   ...
 </table>
```

❶ the button still issues a `POST` to `/integrations/1`

❷ the table no longer listens for an event, but it now has an id

Now let's look at a potential response to the `POST` to `/integrations/1`. This response will include the "regular" content that needs to be swapped into the button, per the usual htmx mechanism. But it will also include a new version, updated version of the contacts table, which will be marked as `hx-swap-oob="true"`. This content will be removed from the response so it is not inserted into the button, but will be instead swapped into the DOM in place of the existing table since it has the same id value.

Listing 11. 145. A Response With Out of Band Content

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...

Pull Contacts From Integration ❶

<table id="contacts-table" hx-swap-oob="true"> ❷
  ...
</table>
```

❶ this content will be placed in the button

❷ this content will be removed from the response and swapped by id

Using this technique, you are able to piggyback content updates of other elements on top of requests by other elements. The `hx-swap-oob` attribute supports other additional features, all of which are documented here: https://htmx.org/attributes/hx-swap-oob/

Depending on how exactly your server side templating technology works, and what level of interactivity your application requires, out of band swapping can be a powerful mechanism for more flexible content updates.

13.4.3. Events

Finally, the most complex mechanism for updating content is the one we saw back in the events section: using server-triggered events to update elements. This approach can be very clean, but also requires a lot deeper conceptual knowledge of HTML and events, and a commitment to the event-driven approach. While we like this style of development, it isn't for everyone and we typically recommend this only if the htmx philosophy of event-driven hypermedia really speaks to you.

If it *does* speak to you, however, we say: go for it! We've created some very complex and flexible user interfaces using this approach, and we are quite fond of it.

13.4.4. Being Pragmatic

All of these approaches to the "Updating Other Content" problem will work, and will often work well. However, there may come a point where it would just be simpler to use a different approach, like the reactive one. As much as we like the hypermedia approach, the reality is that there are some UX patterns that simply cannot be implemented easily using it. The canonical example of this sort of pattern, which we have mentioned before, is something like a live online spreadsheet: it is simply to complex a user interface, with too many inter-dependencies, to be done well via exchanges of hypermedia with a server.

In cases like this, and any time you feel like an htmx-based solution is proving to be more complex than another approach might be, we can gladly recommend that you consider a different technology: use the right tool for the job! You can always use htmx for the parts of your application that aren't as complex and don't need the full complexity of a reactive framework, and save that complexity budget for the parts that do.

We are not hypermedia puritans and encourage you to learn many different web technologies, with an eye to the strengths and weaknesses of each one. This will give you a deep tool chest to reach into when problems present themselves. Our hope is that, with htmx, hypermedia might be a tool you reach for more frequently!

13.5. Debugging

We have been talking a lot about events in this chapter and we are not ashamed to admit: we are big fans of events. They are the underlying technology of almost any interesting user interface, and are particularly useful in the DOM once they have been unlocked for general using in HTML. They let you build nicely decoupled software while often preserving the locality of behavior we like so much.

However, events are not perfect. One area where events can be particularly tricky to deal with is *debugging*: you often want to know why an event *isn't* happening. But where can you set a break point for something that *isn't* happening? The answer, as of right now, is: you can't.

There are two techniques that can help in this regard, one provided by htmx, the other provided by Chrome, the browser by google.

13.5.1. Logging htmx Events

The first technique, provided by htmx itself, is to call the `htmx.logAll()` method. When you do this, htmx will log all the internal events that occur as it goes about its business, loading up content, responding to events and so forth.

This can be overwhelming, but with judicious filtering can help you zero in on a problem. Here are what (a bit of) the logs look like when clicking on the "docs" link on https://htmx.org, with `logAll()` enabled:

Listing 11. 146. htmx Logs

```
htmx:configRequest
<a href="/docs/">
Object { parameters: {}, unfilteredParameters: {}, headers: {…}, target: body, verb:
"get", errors: [], withCredentials: false, timeout: 0, path: "/docs/",
triggeringEvent: a
, … }
htmx.js:439:29
htmx:beforeRequest
<a href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {}, pathInfo:
{…}, elt: a
 }
htmx.js:439:29
htmx:beforeSend
<a class="htmx-request" href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {}, pathInfo:
{…}, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:loadstart
<a class="htmx-request" href="/docs/">
Object { lengthComputable: false, loaded: 0, total: 0, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:progress
<a class="htmx-request" href="/docs/">
Object { lengthComputable: true, loaded: 4096, total: 19915, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:progress
<a class="htmx-request" href="/docs/">
Object { lengthComputable: true, loaded: 19915, total: 19915, elt: a.htmx-request
 }
htmx.js:439:29
htmx:beforeOnLoad
<a class="htmx-request" href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {}, pathInfo:
{…}, elt: a.htmx-request
 }
htmx.js:439:29
htmx:beforeSwap
<body hx-ext="class-tools, preload">
```

Yikes! Not exactly easy on the eyes, is it? But, if you take a deep breath and squint, you can see that it isn't *that* bad: a series of htmx events, some of which we have seen before (there's `htmx:configRequest`!), get logged to the console, along with the element they are triggered on. After a bit of reading and filtering, you will be able to make sense of the event stream, and it can help you debug htmx-related issues.

13.5.2. Monitoring Events in Chrome

The preceding technique is useful if the problem is occuring somewhere *within* htmx, but what if htmx is never getting triggered at all? This comes up some times, like when, for example, you have accidentally typed an event name incorrectly somewhere.

In cases like this you will need recourse to a tool available in the browser itself. Fortunately, the Chrome browser by Google provides a very useful function, `monitorEvents()`, that allows you to monitor *all* events that are triggered on an element. This feature is available *only* in the console, so you can't use it in code on your page. But, if you are working with htmx in Chrome, and are curious why an event isn't triggering on an element, you can open the developers console and type the following:

Listing 11. 147. htmx Logs

```
monitorEvents(document.getElementById("some-element"));
```

This will then print *all* the events that are triggered on the element with the id `some-element` to the console. This can be very useful for understanding exactly which events you want to respond to with htmx, or troubleshooting why an expected event isn't occurring.

Using these two techniques will help you as you (infrequently, we hope!) troubleshoot event-related issues when developing with htmx.

13.6. Security Considerations

In general, htmx and hypermedia tends to be more secure than JavaScript heavy approaches to building web applications. This is because, by moving much of the processing to the back end, the hypermedia approach tends not to expose as much surface area of your system to end users for manipulation and shenanigans.

However, even with hypermedia, there are still situations that require care when doing

development. Of particular concern are situations where user-generated content is shown to other users: a clever user might try to insert htmx code that tricks the other users into clicking on content that triggers actions they don't want to take.

In general, all user-generated content should be escaped on the server side, and most server side rendering frameworks provide functionality for handling this situation. But there is always a risk that something slips through the cracks.

In order to help you sleep better at night, htmx provides the `hx-disable` attribute. When this attribute is placed on an element, all htmx attributes within that element will be ignored.

13.6.1. Content Security Policies & htmx

A Content Security Policy (CSP) is a browser technology that allows you to detect and prevent certain types of content injection-based attacks. A full discussion of CSPs is beyond the scope of this book, but we refer you to the Mozilla Developer Network article on them for more information: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

A common feature to disable using a CSP is the `eval()` feature of JavaScript, which allows you to evaluate arbitrary javascript code from a string. This has proven to be a security issue and many teams have decided that it is not worth the risk to keep it enabled in their web applications.

htmx does not make heavy use of `eval()` and, thus, a CSP with this restriction in place will be fine. The one feature that does rely on `eval()` is event filters, discussed above. If you decide to disable `eval()` for your web application, you will not be able to use the event filtering syntax.

13.7. Configuring

There are a large number of configuration options available for htmx. Some examples of things you can configure are:

- The default swap style
- The default swap delay
- The default timeout of AJAX requests

A full list of configuration options can be found in the config section of the main htmx

documentation: https://htmx.org/docs/#config

htmx is typically configured via a `meta` tag, found in the header of a page. The name of the meta tag should be `htmx-config`, and the content attribute should contain the configuration overrides, formatted as JSON. Here is an example:

Listing 11. 148. An htmx configuration via a `meta` tag

```
<meta name="htmx-config" content='{"defaultSwapStyle":"outerHTML"}'>
```

In this case, we are overriding the default swap style from the usual `innerHTML` to `outerHTML`. This might be useful if you find yourself using `outerHTML` more frequently than `innerHTML` and want to avoid having to explicitly set that swap value throughout your application.

13.8. Summary

- In this chapter we looked at some details and tricks of htmx development
- We looked in detail at the options available for `hx-swap` and `hx-trigger`, including filters and scrolling
- We took a look at the events that htmx triggers and responds to
- We explored HTTP response headers and HTTP response codes in htmx, and how to modify how htmx handles them
- We looked at various techniques for updating content beyond the target of a request, including out of band swaps
- We saw how htmx applications can be debugged, secured and configured = Hypermedia In Action :chapter: 12 :sectnums: :figure-caption: Figure 11. :listing-caption: Listing 11. :table-caption: Table 11. :sectnumoffset: 11 :leveloffset: 1 :sourcedir: ../code/src :source-language:

14. Other Hypermedia Libraries & Technologies

This chapter covers

- Hypermedia libraries beyond htmx and HyperView

- Non-HTML hypermedia technologies

Non-htmx Hypermedia Oriented Libraries

As the creators of htmx, we are obviously partial to its approach to building Hypermedia Driven Application. We like how it builds incrementally on top of existing concepts within HTML and stays "close to the metal" in that regard. However, it is far from the only library that takes hypermedia seriously as a technology. There are other excellent libraries that take different approaches, some similar some very different, while at the same time still respecting the core concept of the web: exchanging HTML with the server.

In addition to these HTML-based technologies, there is a whole world of non-HTML hypermedias. We have covered one in depth in this book: HyperView, a mobile hypermedia. But it is not the only non-HTML hypermedia out there, and it behooves us to look at some of these other technologies as well.

14.1. Unpoly

Unpoly is a mature and stable hypermedia-oriented library that has been in existence for nearly a decade. It came out of the Rails community in the 2010s, which was a hot-bed for innovation in the web application world at that time.

Unpoly provides quite a bit more functionality than htmx does. For example, it supports:

- A wide array of animations for transitions, such as sliding
- A notion of "layers", allowing for native modal-like behavior
- Built in preloading of content (in htmx this requires an extension)
- A strong focus on progressive enhancement, so non-JS enabled clients continue to work

Like htmx, Unpoly uses attributes on elements to add enhancements beyond their normal behavior. However, unlike htmx, Unpoly strongly encourages the use of standard anchors and forms for hypermedia interactions. This is in keeping with its focus on graceful degredation when JavaScript is not available.

Here is some sample Unpoly code, to give you a flavor of the library:

Listing 11. 149. An Unpoly Powered Link

```
<a href="/contacts" up-follow
                    up-target=".contacts-table">
  Get Contacts
</a>
```

Here you can see the `up-follow` attribute, which tells Unpoly "when a user clicks this link, issue the request as an AJAX request instead". You can also see the `up-target` attribute, which, like the `hx-target` attribute in htmx, allows you to specify the target element for replacement.

Like the `hx-boost` attribute we looked at earlier, Unpoly will update the navigation bar with the new URL, updating history.

14.1.1. Layers

Unpoly has a concept of layers, which allows you to open new content in an overlay. This allows for a more sophisticated notion of windowing in your web application that is available via the normal DOM API. Layers can be shown as modal dialogs, as a drawer that slides out from the side, as a popup anchored to a given link or as an overlay, in addition to the normal "root" layer, where content will replace the current window.

As an example, here is a screenshot from the Unpoly demo application, available at https://demo.unpoly.com

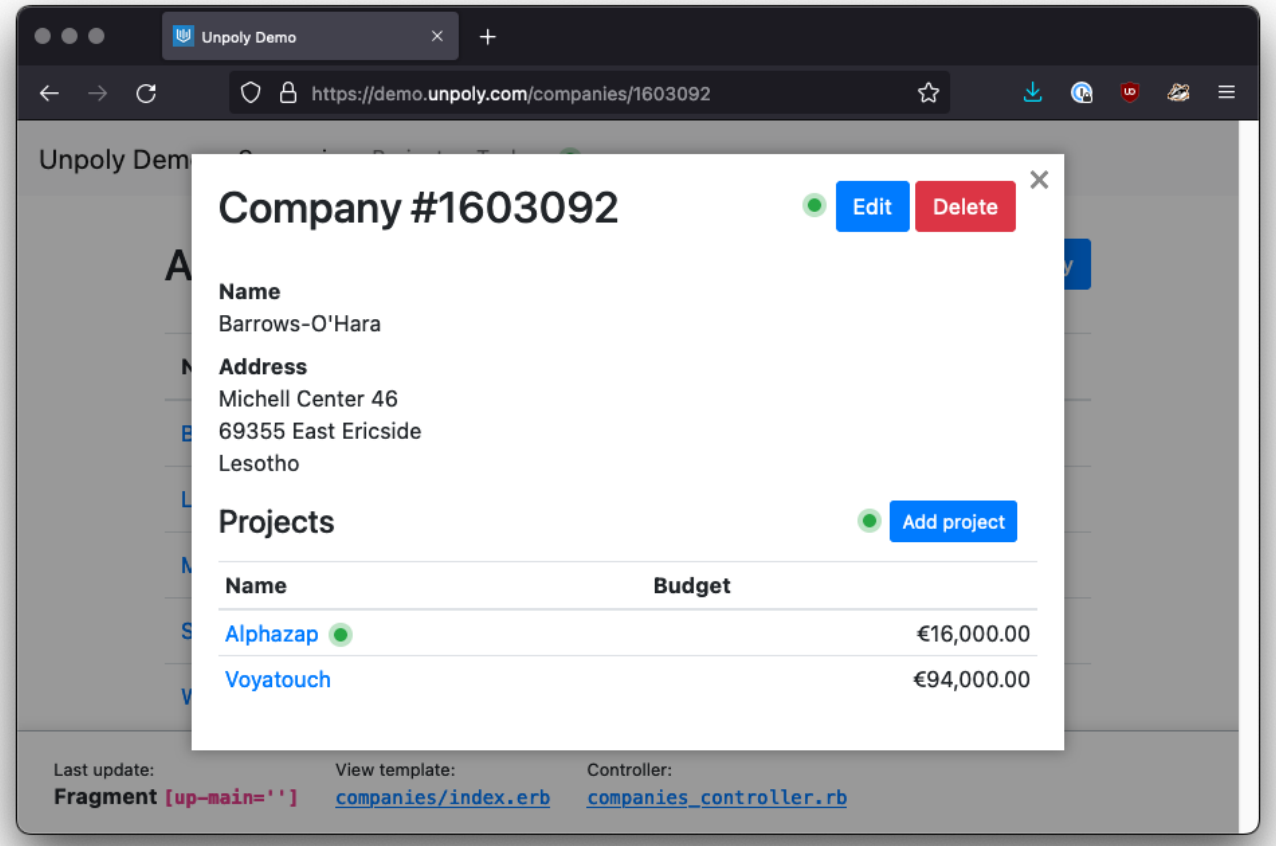


Figure 11. 10. An Unpoly Modal

This modal was launched by clicking on a normal anchor tag link that had been annotated to indicate that the result should be placed in a new modal layer. Unpoly provided the infrastructure to show the response from the server within a modal window, rather than replacing the entire screen, as well as dismiss the window. And a really nice feature here is that, if the user does not have JavaScript enabled, the web application continues to work, since the link will simply follow the normal "replace the entire screen" logic of web applications.

14.1.2. Unpoly's JavaScript API

Unpoly ensures that all functionality that is available via attributes is also available via JavaScript, allowing complete access to the underlying concepts and ideas. This makes Unpoly a great library to pair with JavaScript scripting in your Hypermedia Driven Application.

Given the link above to retrieve `/contacts`, you can write the following code:

Listing 11. 150. Triggering An Unpoly Powered Link Via JavaScript

```
var link = document.querySelector('a')
up.follow(link)
```

And Unpoly will load the link just like if the user had clicked on it.

Unpoly is a great option to consider when building a Hypermedia Driven Application. It provides more structure than htmx does, and focuses on progressive enhancement. If these factors are important to you, Unpoly might be a better choice that htmx. htmx, on the other hand, is lower level and stays closer to HTML conceptually.

14.2. Hotwire

Hotwire is a collection of libraries produced by 37 Signals, the company that also created the Ruby on Rails web framework. Hotwire stands for "HTML Over The Wire", and consists of three different components:

- Turbo - A library for building HTML-driven web applications
- Stimulus - A library for writing JavaScript within a Turbo-based web application
- Strada - A mobile library for integrating with HTML-drive web applications

Turbo is the core library of Hotwire, and, like Unpoly, it offers a lot more structure than htmx does. Turbo has the concept of "turbo frames", which are pre-defined areas that can be updated independently from one another. To take the Unpoly example above and convert it to Turbo, we would remove the direct annotations on the anchor tag and instead wrap it in a `turbo-frame` element, which is a custom HTML element provided by the Turbo library:

Listing 11. 151. An Turbo Powered Link

```
<turbo-frame id="contacts-table">
     ...
    <a href="/contacts">
      Get Contacts
    </a>
    ...
</turbo-frame>
```

When a user clicks on this link, the returned content will be inserted into the turbo-frame

appropriately.

14.2.1. Web Sockets

A unique feature of Turbo is that it can use Web Sockets for exchanging hypermedia with a server. Web Sockets are a browser technology that allows for a persistent connection to be maintained with a server. By maintaining a persistent connection, you can avoid the set up and tear down costs associated with establishing an HTTP connection, and you also allow the server to "push" messages down to the client.

Turbo supports a `turbo-stream` element that can be tied to a Web Socket, allowing you to stream HTML down to the client. This can make for a very dynamic user interface and works well for things like an online messaging application. (Hotwire was designed in large part to support the Hey! online email application, an email client that has many instant messaging-like features.)

Hotwire would be a particularly good choice for your Hypermedia Driven Application if you are using Ruby on Rails as your back end server: since both technologies are developed by the same people, they work seamlessly with one another. Hotwire can be used with other back end technologies, as well. As with Unpoly, Hotwire gives you more structure than htmx does, with a notion of "frames" and "streams". Hotwire also supports streaming HTML over Web Sockets, something that htmx supports only secondarily.

Hotwire also has the advantage that it is supported by a mature and famous company, who depends on it for their own applications.

14.3. jQuery

You may be surprised to see jQuery on this list. jQuery is the granddaddy of them all of JavaScript libraries, and is where many older developers first started exploring AJAX, using the `ajax()` function. jQuery is typically referred to by the somewhat cryptic `$` symbol.

Let’s take the example "Get Contacts" link that we have been using and convert it to jQuery. To do so we are going to have to get a bit into the weeds with respect to how jQuery works.

When you are writing jQuery code, you tend to separate out your JavaScript from your HTML, a concept we discussed in Chapter 9 called "Separation of Concerns". The general

pattern is: in the "ready" event for the page (that is, when the page is fully rendered), look up elements in the page and wire in event handlers.

In our case, we are going to look up the "Get Contacts" element, which we will change to a button, since we don't want the link behavior anymore. We do this using the jQuery query syntax, which is to pass a CSS selector into the `$()` function. Don't worry too much about the details if you aren't familiar with jQuery, the important thing to understand conceptually is that we "look up" the button and then wire in a click handler.

From the click handler, we can invoke the `ajax()` method in jQuery. We can then take the result and put it into another element by looking that element up and calling the `html()` function with the HTML content that is return by the server.

Here is what our code might look like:

Listing 11. 152. Issuing an AJAX Request in jQuery

```
<script>
    $(document).ready(function() {
        $("#contacts-btn").click(function() {
            $.ajax({url: "/contacts",
                    success: function(result) {
                $(".contacts-table").html(result);
            }});
        });
    });
</script>
<button id="contacts-btn">
  Get Contacts
</button>
```

Not the cleanest looking code in the world, at least to our eyes, but it works, and it is using hypermedia: we are not exchanging JSON with the server here. We are, instead, pulling down HTML and placing it in the DOM, just like htmx does.

This style of jQuery code, in fact, was the early inspiration of intercooler.js, the predecessor to htmx. intercooler.js was built on top of jQuery, initially as a jQuery extension and then as a stand-alone library. htmx was created as "intercooler 2.0" and the jQuery dependency was removed.

14.4. VanillaJS

Although still widely used, jQuery has become less popular over time as JavaScript has standardized across browsers, and as the native JavaScript APIs have improved. (These improvements have often been inspired by jQuery!) Today many people would prefer to use plain JavaScript.

It turns out that using plain JavaScript to implement a simple Hypermedia Driven Application is pretty easy today, thanks to the addition of the `fetch()` API! The `fetch()` API allows you to issue AJAX requests in a much simpler manner than the older `XMLHttpRequest` API: you can simply call a function, `fetch()`, with the URL that you want to issue the HTTP request to.

Let's rework the jQuery button above to use `fetch()`. We will inline the code directly on the button using an `onclick` handler, which is more in line with how people use JavaScript today:

Listing 11. 153. Issuing an AJAX Request using `fetch()`

```
<button onclick="fetch('/contacts')
                  .then((response) => response.text())
                  .then((data) =>{
                      document.querySelector('.contacts-table').innerHTML = data;
                  };">
  Get Contacts
</button>
```

As with the jQuery example, not exactly easy on the eyes, at least in our opinion, but it works! We call the `fetch()` function, passing in the path that we want to issue the AJAX request to, then convert the response to text, then look up the contact table by a class name, and then set its inner HTML to the content that came back from the server.

I hope you can see why we were inspired to create htmx, based on this code: there is an awful lot of syntax to achieve a relatively simple end goal of pulling down some HTML and loading it into the DOM. Nonetheless, if you only need to do this style of AJAX request in a few spots in your web application, the fact that you don't need any additional libraries using this approach may outweigh the syntactic convenience and additional functionality that htmx provides.

14.5. Non-HTML Hypermedia

We have focused on two hypermedia formats in this book: HTML and HyperView. These two hypermedia formats address the two most common platforms for building online applications: the browser and mobile clients. HTML is the most widely known and deployed hypermedia in the world, whereas HyperView is a relatively new and unknown technology. But are there other hypermedia formats out there?

It turns out that yes, there are.

14.5.1. Atom

Atom is an XML-based hypermedia format that support publishing and editing web resources. It is designed to help websites publish a feed of updated content in a standard hypermedia format. It was developed as an alternative to the Real Simple Syndication (RSS) format, an updated format that resolved some of the problems with that earlier format.

After HTML, Atom is probably the most widely deployed hypermedia format. It became very popular for news and blog-style websites in the late 2000s.

Atom explicitly includes hypermedia controls, the ability to edit resources and follow urls, via the `link` element, which can include a `rel` attribute (short for "relation") indicating exactly what the links relationship to the given representation is.

Here is an example Atom document:

Listing 11. 154. An Example Atom Document

```
<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
    <title>Atom-Powered Robots Run Amok</title>
    <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
    <updated>2003-12-13T18:30:02Z</updated>
    <author><name>John Doe</name></author>
    <content>Some text.</content>
    <link rel="edit" href="http://example.org/edit/first-post.atom"/> ❶
</entry>
```

❶ A hypermedia control, telling an atom client where this entry can be updated

As you can see, Atom looks, to an extent, like HTML, but HTML tailored to represent a feed of articles. The `link` element within the document is a hypermedia control, telling an Atom client where this resource can be updated via either a `PUT` or `DELETE`.

The Atom hypermedia format might be useful for you if you are, for example, creating a CMS or blogging system, and wish to expose a feed using a hypermedia format.

14.5.2. Hypertext Application Language (HAL)

The Hypertext Application Language (HAL) is another non-HTML hypermedia. Interestingly, this hypermedia supports *both* XML and JSON. Since most of the JSON APIs we have been looking at in this book have been Data APIs, let's take a look at how HAL imposes a hypermedia on top of that format.

In its JSON form, HAL consists of a root JSON object with two reserved "meta-properties":

- `_links` - contains links to other resources
- `_embedded` - contains embedded resources

The `_links` property in JSON responses provides hypermedia control information for the current resource. For a collection of contacts, here is what a `_links` property might look like in a HAL response:

Listing 11. 155. A HAL `_links` property

```
"_links": {
  "self": { "href": "/contacts" },
  "next": { "href": "/contacts?page=2" },
  "find": { "href": "/contacts{?id}", "templated": true }
```

The `_embedded` property is used to enclose other resources, such as a collection of contacts with their own HAL controls. In the case of our collection of contacts, this would include the details of each contact, as well as nested `_links` and/or `_embedded` entries within them.

If we were to add HAL support to the contacts Data API we built in Chapter 9, we woudl add the `_links` property as a top level property in our response, and move the contacts property inside of the embedded property. Our response might look something like this:

Listing 11. 156. Updating our Contacts Data API to include HAL controls

```
{
  "_links": {
    "self": {
      "href": "/contacts"
    },
    "next": {
      "href": "/contacts?page=2"
    },
    "find": {
      "href": "/contacts{?id}",
      "templated": true
    },
    "_embedded": {
      "contacts": [
        {
          "_links": {
            "self": {
              "href": "/contacts/2"
            }
          },
          "email": "carson@example.comz",
          "errors": {},
          "first": "Carson",
          "id": 2,
          "last": "Gross",
          "phone": "123-456-7890"
        }
      ]
    }
  }
}
```

Our JSON API now includes hypermedia controls! The top level `_links` property includes links to both the current and next page of contacts. And, within the `_embedded` property, you can see nested `_links` properties for the contacts, with a "self" link to the URL for the given, embedded contact. Very cool!

This approach to building a JSON API exposes a general, uniform API to the outside world, and a proper hypermedia client will be able to work with these resources without any deep understanding of the internal meaning of the data.

14.5.3. Creating Your Own Hypermedia Format

Another approach, one championed by Mark Amundsen, is to build your *own* hypermedia. This may sound a bit crazy at first, but Mark has written a series of books on exactly this topic. One, in particular, that we can recommend is "Restful Web Clients", which goes over the design of both good hypermedia APIs as well as how to write good hypermedia **clients** for those APIs.

In our experience, the idea of hypermedia APIs are fairly clear to most developers. However, understanding how to create a proper hypermedia **client** is a tricky, and much less discussed part of the hypermedia puzzle. One of the reasons we are so excited about HyperView, and were thrilled to have Adam Stepinski, the creator of HyperView, join us as a co-author of this book, is because he did the hard work of not only defining a hypermedia format, but also of creating a hypermedia client that can work with that format. By creating both sides of the hypermedia puzzle for mobile application, Adam has made HyperView a far more practical and useful technology!

14.5.4. Summary

- In this book we've looked at two hypermedia-based technologies in depth: htmx and HyperView
- There are other hypermedia-oriented libraries out there worth considering, especially for web development. Unpoly and Hotwire are two popular ones.
- There are hypermedia beyond HTML, such as Atom, a hypermedia for representing feeds of articles and HAL, as simple hypermedia format for enhancing your JSON APIs with hypermedia controls.
- If you are ambitious, you might even consider creating your own hypermedia format and client! = Hypermedia In Action :chapter: 13 :sectnums: :figure-caption: Figure 11. :listing-caption: Listing 11. :table-caption: Table 11. :sectnumoffset: 12 :leveloffset: 1 :sourcedir: ../code/src :source-language:

15. Conclusion

Hypermedia Reconsidered

I hope that in this book we have managed to convince you that hypermedia, rather than being a "legacy" technology or a technology only appropriate for "documents" of links, text

and pictures, is, in fact, a powerful technology for building *applications*. In the books have seen how to build sophisticated user interfaces for both the web, with htmx, and for a mobile application, using HyperView, using hypermedia as a core underlying application technology.

Focusing in on the web, in particular, many developers today view the links and forms of "plain" HTML as bygone tools from a less sophisticated age. And, in some ways, they are right: there were definite usability issues with the original web. However, as we discovered while building Contact.app, by simply addressing four core limitations of HTML:

- By making any element capable of issuing an HTTP request
- By making any event capable of triggering an HTTP event
- By making all the different types of HTTP actions available
- By aking it possible to target any element in the DOM for replacement

We were able to build user interfaces that many developers would assume require a significant amount of client-side JavaScript, but using only hypermedia concepts.

The Hypermedia Driven Application approach is not right for every application (after all, no approach is right for *every* application) but, for many applications, the increased flexibility and simplicity of hypermedia can be a huge benefit. And, even if your application wouldn't benefit from this approach, it is at least worthwhile to *understand* the approach, it's strengths and weaknesses, and how it differs from the approach you are taking. The original web grew faster than any distributed system in history, and it is worth understanding the underlying technologies that made that growth possible.

15.1. Pausing, And Reflecting

The JavaScript community and, by extension, the web development community is famously chaotic, with new frameworks and technologies emerging monthly, and sometimes even *weekly*. It can be exhausting to keep up with the latest and greatest technologies that are coming out, and, at the same time, terrifying that we *won't* keep up with them and be left behind in our career.

This is not a fear without foundation: there are many senior software engineers that have seen their career peter out because they picked a technology to specialize in that, fairly or not, did not thrive. The web development world tends to be young, with many companies

favoring young developers over older developers who "haven't kept up."

We shouldn't sugar-coat these realities of our industry. On the other hand, we also shouldn't ignore the downside that these realities create. It creates a high-pressure environment where everyone is watching for "the new new" thing, that is, for the latest and greatest technology that is going to change everything. It creates pressure to *claim* that your technology is going to change everything, if you are trying to get attention. It tends to favor *sophistication* over *simplicity*. People are scared to ask "Is this too complex?" because it sounds an awful lot like "I'm not smart enough to understand this."

The software industry has tended, especially in web development, to lean far more towards innovating, rather than understanding the technologies of the past and building on them or within them. We tend to look ahead for new, genius solutions, rather than looking backwards to older ideas. This is understandable: the technology world is necessarily a forward-looking industry.

On the other hand, there have been a lot of great ideas in the past, many of which have been discarded. We are old enough to have seen hypermedia come and go as the "new new" idea. It was a little shocking to us to see it discarded so cavalierly by the industry. Fortunately, the concepts are still sitting there, waiting to be discovered and reinvigorated by a new generation of web developers. The original, RESTful architecture of the web, when looked at with fresh eyes, can address many of the problems todays web developers are facing.

Perhaps, following Mark Twain's advice, it is time to pause and reflect. Perhaps, for a few quiet moments, we can put the endless swirl of the "new new" aside, look back on where the web came from, and learn.

Perhaps it's time to give hypermedia a chance.

I hope you will.

16. Hypermedia In Action

17. REST, HATEOAS and All That

This appendix covers

- An in-depth look at hypermedia, in terms of HTML and HTTP

- Representational State Transfer (REST)
- Using Hypermedia As The Engine of Application State (HATEOAS)

17.1. Hypermedia, HTML & HTTP: A In-depth Exploration

Hypermedia is a non-linear medium of information that includes various sorts of media such as images, video, text and, crucially, hyperlinks: references to other data.

Hypertext is a subset of hypermedia that consists of *text* that supports *hypermedia controls* being embedded in it.

The most common hypertext today is the HyperText Markup Language (HTML), and the most common hypermedia controls are anchor tags and form tags.

17.1.1. Hyperlinks

Hyperlinks in HTML are created via anchor tags, and they specify their references to other data (or *resources*) via Universal Resource Locators, or URLs. A URL looks like this:

```
https://www.manning.com/books/hypermedia-in-action
```

A URL typically consists of at least the following components:

- A protocol or scheme (in this case `https`)
- A domain (in this case `www.manning.com`)
- A path (in this case `/books/hypermedia-in-action`)

A URL uniquely identifies a retrievable *resource* on the internet.

A web browser will turn an anchor tag found in HTML into a visually distinct bit of text that, when clicked on, will cause the browser to issue a HyperText Transfer Protocol (HTTP) network request to the URL specified in the anchor.

Consider this small fragment of HTML:

```
<a href="/contacts/42">Joe Smith</a>
```

When a user clicks on this anchor, rendered as a hyperlink in a browser, an HTTP request

will be issued by the browser that looks something like this:

```
GET http://example.org/contacts/42 HTTP/1.1
Accept: text/html,*/*
Host: example.org
```

The first line specifies that this is an HTTP `GET` request. It then specifies the path of the resource being requested. Finally, it contains the HTTP version for this request.

After that are a set of HTTP *Request Headers*, individual lines of name/value pairs, separated by a colon, which provide *metadata* that can be used by the server to determine exactly how to respond to the client request. In this case, the client is saying it would prefer HTML as a response format, but will accept anything.

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

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 870
Server: Werkzeug/2.0.2 Python/3.8.10
Date: Sat, 23 Apr 2022 18:27:55 GMT

<html lang="en">
<head>
<h1>Joe Smith</h1>
<div>
    <div>Phone: 123-456-7890</div>
    <div>Email: joe@example.bar</div>
</div>
<p>
    <a href="/contacts/42/email">Email Joe Smith</a>
</p>
</main>
</body>
</html>
```

Here the HTTP Response specifies a *Response Code* of `200`, indicating that the given resource was found, and the request succeeded.

As with the HTTP Request, we see a series of *Response Headers* that provide metadata to

the client to assist in displaying the *representation* of the resource correctly.

Finally, we see some new HTML content. This content is the HTML *representation* of the requested resource. The browser will use to replace the entire content in its display window, showing the user a new page and, typically, updating the address bar to reflect the new URL.

17.1.2. Forms

HTML also supports the form tag hypermedia control. Forms allow a user to interact with servers in a richer manner than anchors do. Form tags can submit either `GET` requests or `POST` HTTP requests. We will discuss these two actions in more detail in a bit.

A simple form tag looks like this:

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

This form will be rendered as simple text input, with a button next to it. If a user enters the value `example@example.org` in the email input and then submits the form (either by clicking on the button or by hitting the enter key while the text input is focused). the form will issue a `POST` request that looks something like this:

```
POST http://example.org/signup HTTP/1.1
Accept: text/html,*/*
Host: example.org

email=example%40example.org
```

The first line specifies that this is an HTTP `POST` request. It then specifies the path of the resource being posted to, and finally it is followed by the HTTP version for this request.

As with the hyperlink example, we see some HTTP request headers.

After these headers, we see something new: a request *body*. This body carries the information that is being posted to the server, using form-url encoding. (That's why there is a funny `%40`, taking the place of the `@` symbol in the email that was submitted.)

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

```
HTTP/1.1 301 Moved Permanently
Location: https://www.example.org/thank-you
Content-Type: text/html

<html>
<head>
<title>Moved</title>
</head>
<body>
<h1>Moved</h1>
<p>This page has moved to <a href="https://www.example.org/thank-
you">https://www.example.org/thank-you</a>.</p>
</body>
</html>
```

This response uses the `301` HTTP Response code, which tells the browser "This page is not the final URL for the response to this request, rather issue a `GET` to `https://www.example.org/thank-you`, which will give you the final content."

The browser will then issue a `GET` request to this new URL and load the content returned by it into the browser window, presumably a "Thank you for signing up" page.

This is a simple example of the widely used *Post/Redirect/Get* pattern from the early web. By adopting this pattern of redirection after a `POST` occurs, the `POST` request and response does not end up in the browser history. This means that if the user hits the "Refresh" button, the `POST` is not issued. Rather, the browser will issue a `GET` to the final URL is that it navigated to. This avoids accidentally re-updating a resource by simply refreshing a page.

If you have ever seen a warning by a browser saying something like "Are you sure you wish to refresh this page?" it is most likely because the website you are using is not properly using this Post/Redirect/Get pattern.

17.1.3. HTTP Methods

It turns out that the HTTP protocol supports a number of request methods or verbs, not just `GET` and `POST`. The most relevant methods for web application developers are as follows:

`GET`	A GET request requests the representation of the specified resource. GET requests should not mutate data.
`POST`	A POST request submits data to the specified resource. This will often result in a mutation of state on the server.
`PUT`	A PUT request replaces the data of the specified resource. This results in a mutation of state on the server.
`PATCH`	A PUT request replaces the data of the specified resource. This results in a mutation of state on the server.
`DELETE`	A DELETE request deletes the specified resource. This results in a mutation of state on the server.

These verbs roughly line up with the "Create/Read/Update/Delete" or CRUD pattern in development:

- `POST` corresponds with Create
- `GET` corresponds with Read
- `PUT` and `PATCH` correspond with Update
- `DELETE` corresponds, well, with Delete

In a properly structured hypermedia system, you should use the appropriate HTTP method for the operation a given element performs: If an element such as a button *deletes* a resource, for example, ideally it should use the `DELETE` method.

HTML & HTTP Methods

A funny thing about HTML is that, despite being the world's most popular hypermedia and despite being designed alongside HTTP (which is the Hypertext Transfer Protocol, after all), HTTP can only issue `GET` and `POST` requests directly.

Anchor tags always issue a `GET` request.

Forms can issue either a `GET` or `POST` using the `method` attribute.

But forms and anchor tags can't issue `PUT`, `PATCH` or `DELETE` requests! If you wish to issue these last three types of requests, you currently have to resort to JavaScript.

This is an obvious shortcoming of HTML as a hypermedia, and it is hard to understand why this hasn't been fixed in the HTML specification yet!.

17.2. REpresentational State Transfer (REST)

So, brief refresher on hypermedia, HTML & HTTP out of the way, we are ready to take a close look at the concept of REST. The term REST comes from Chapter 5 of Roy Fielding's PhD dissertation on the architecture of the web. Fielding wrote his dissertation at U.C. Irvine, after having helped build much of the infrastructure of the early web, including the apache web server. Roy was attempting to formalize and describe the novel distributed computing system he had just helped to build.

We are going to focus in on what I feel is the most important section, from a web development perspective: section 5.1. This section contains the core concepts (Fielding calls them *constraints*) of Representational State Transfer, or REST.

Before we get into the details, however, it is important to understand that Fielding considers REST a *network architecture*, that is an entirely different way of architecting a distributed system, that should be contrasted with earlier distributed systems.

REST was and is not simply a checklist of TODOs for an API end point within a broader application. It is, rather, a unique network architecture for an entire system. It needs to be understood *conceptually*, rather than as a rote list of things to tick off as you develop a particular system HTTP end point.

It is also important to emphasize that, at the time Fielding wrote his dissertation, JSON APIs and AJAX *did not exist*. He was *describing* the early web, HTML being transferred over HTTP, as a hypermedia system.

Today the term "REST" is mainly associated with JSON APIs. We feel this term is typically used erroneously when discussing these APIs, which are much better described as *Data APIs*. We will discuss the difference between these Data APIs and a truly REST-ful system in depth below.

Again: REST describes *the pre-JSON API web*, and letting go of the current common usage of the term "REST" is necessary to develop a proper understanding of it conceptually.

17.2.1. The "Constraints" of REST

Fielding uses various "constraints" to describe how a REST-ful system must behave. This approach is a little round-about and academic, but once you spend some time thinking about these constraints, it should become easy to understand if a system actually satisfies these architectural requirements of REST or not.

Here are the constraints of REST, from Section 5.1 of his dissertation:

- It is a client-server architecture (section 5.1.2)
- It is stateless (section 5.1.3) that is, every request contains all information necessary to respond to that request; no side state is maintained
- It allows for caching (section 5.1.4)
- It consists of a *uniform interface* (section 5.1.5)
- It is a layered system (section 5.1.6)
- Optionally, it allows for Code-On-Demand (section 5.1.7), that is, scripting.

Let’s go through each of these constrains in turn and discuss them in detail.

17.2.2. Client-Server

See https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_2 for the Client-Server constraint.

Obviously, the REST model Fielding was describing involved both *clients* (that is, Web Browsers) and *servers* (such as the Apache Web Server he had been working on)

communicating via a network connection. This was the context of his work: he was describing the **network architecture** of the World Wide Web, and contrasting it with earlier architectures, notably thick-client networking models such as the Common Object Request Broker Architecture (CORBA).

It should be obvious that any web application, regardless of how it is designed, is going to satisfy this requirement.

17.2.3. Statelessness

See https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_3 for the Stateless constraint.

As described by Fielding, a REST-ful system is stateless: every request should encapsulate all information necessary to respond to that request, with no side state or context stored on the server.

In practice, for many web applications today, we actually violate this constraint: it is common to establish a *session cookie* that acts as a unique identifier for a given user and that is sent along with every request. While this session cookie is, by itself, not stateful (after all, it is sent up with every request), it is typically used as a key to look up information stored on the server side, in what is usually termed "the session". This session information is typically stored in some sort shared storage across multiple web servers, and holds things like the current users email or id, their roles, partially created domain objects, catches, and so forth.

This violation of the Statelessness REST architectural constraints has proven to be useful for building web applications and, for the most part, does not appear to have had a significant impact on the overall flexibility of the approach.

It should be noted, however, that it does cause additional complexity headaches when deploying hypermedia servers, which now may need to have shared access to the session state information, for example.

17.2.4. Caching

See https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_4 for the Cache constraint.

This constraint states that a RESTful system should support the notion of caching, with

explicit information on the cacheability of requests for future requests of the same resource.

HTTP has an extensive caching mechanism that is often under-utilized for web applications. Via the judicious use of HTTP Headers you can ask browsers to keep a response for a given URL in a local cache and, when that URL is requested, reuse that locally cached content.

17.2.5. The Uniform Interface Constraint

Now we come to the most interesting and, in our opinion, innovative constraint in REST: the *uniform interface*. This constraint is the source of much of the *flexibility* and *simplicity* of a hypermedia system, so we are going to spend a lot of time on it.

See https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5 for the Uniform Interface.

In this section, Fielding says:

> The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components… In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state
>
> — Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

So we have four additional sub-constraints that, taken together, form the Uniform Interface constraint.

Identification of Resources

In a REST-ful system, resources should have a unique identifier. Today the concept of Universal Resource Locators (URLs) is common, but at the time of Fielding's writing they were still relatively new and novel.

What might be more interesting today is the notion of a *resource*, thus being identified: in a REST-ful system, *any* sort of data that can be referenced, that is, the target of a hypermedia reference, is considered a resource. URLs, though common enough today, end up solving the very complex problem of uniquely identifying any resource on the internet!

Manipulation of Resources Through Representations

In a REST-ful system, *representations* of the resource are transferred between clients and servers. These representations can contain both data and metadata about the request (control data). A particular data format or *media type* may be used to present a given resource to a client, and that media type can be negotiated between the client and the server.

We saw this latter aspect of the uniform interface in the `Accept` header in the requests above.

Self-Descriptive Messages

This constraint, along with the next one, form what we consider to be the crux of the Uniform Interface, of REST and why, in our opinion, hypermedia is such a powerful network architecture: in a REST-ful system, messages must be *self-describing*.

What does that mean?

It means that messages must contain *all information* necessary to both display *and also operate* on the data being represented.

This might sound a little abstract, so perhaps an example will help to clarify. Consider two implementations of an endpoint, `/contacts/42` both of which return a representation of a Contact.

The first implementation returns an HTML representation:

```
<html lang="en">
<head>
<h1>Joe Smith</h1>
<div>
    <div>Email: joe@example.bar</div>
    <div>Status: Active</div>
</div>
<p>
    <a href="/contacts/42/archive">Archive</a>
</p>
</main>
</body>
</html>
```

The second implementation returns a JSON representation:

```
{
  "name": "Joe Smith",
  "email": "joe@example.org",
  "status": "Active"
}
```

What can we say about the differences between these two responses?

Well, one thing that probably jumps out at you is that the JSON representation is much less verbose than the HTML representation. Fielding notes exactly this tradeoff in hypermedia-based systems in his dissertation:

> The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs.
>
> — Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

So the hypermedia trades off representational efficiency for other goals, and you will often see this leveled as a complaint about HTML: it's just so *verbose* compared to the JSON equivalent.

This is a valid criticism, although we would note that the difference between the two responses is almost certainly a round-off error when compared with network latency, connecting to a server-side data store, and so forth.

But let us grant that the JSON response is better in this regard. In what way is the HTML response better?

Notice that the HTML representation has a hyperlink in it to navigate to a page to archive the contact. The JSON representation, in contrast, does not. What are the ramifications of this fact for a client of the JSON API?

What this means is that the JSON API client **must understand** what the "status" field of a contact means. If the JSON client is able to update the contact in some way, it must know how to do so from some source *external* to the JSON message. This source might be API documentation, word of mouth or, if the developer controls both the server and the client, internal knowledge.

The HTML client, on the other hand, needs only to know how to render HTML. It doesn't

need to understand what the "status" field on a Contact means and, in fact, doesn’t need to understand what a Contact means at all!

It simply renders the HTML and allows the user, who presumably understands the concept of a Contact, to make a decision on what action to pursue.

This difference between the two responses demonstrates the crux of REST and hypermedia, what makes them so powerful and flexible: clients (that is, web browsers) don’t need to understand *anything* about the underlying resources being represented.

They only (only!) need to understand how to parse and display hypermedia, in this case HTML. This gives hypermedia-based systems unprecedented flexibility in dealing with changes to both the backing representations and to the system itself. This will become more apparent as we further explore this idea below.

Hypermedia As The Engine of Application State (HATEOAS)

The final sub-constraint on the Uniform Interface is that, in a REST-ful system, hypermedia should be "the engine of application state". This is sometimes called "HATEOAS", although Fielding prefers "the hypermedia constraint."

This constraint is closely related to the self-describing message constraint. Let us consider again the two different implementations of the end point `/contacts/42`, one returning HTML and one returning JSON. Let’s update the situation such that the contact identified by this URL has now been archived.

What do our responses look like?

The first implementation returns the following HTML:

```
<html lang="en">
<head>
<h1>Joe Smith</h1>
<div>
    <div>Email: joe@example.bar</div>
    <div>Status: Archived</div>
</div>
<p>
    <a href="/contacts/42/unarchive">Unarchive</a>
</p>
</main>
</body>
</html>
```

The second implementation returns the following JSON representation:

```
{
  "name": "Joe Smith",
  "email": "joe@example.org",
  "status": "Archived"
}
```

What to notice here is that, by virtue of being a self-describing message, the HTML response now shows that the "Archive" operation is no longer available, and a new "Unarchive" operation has become available. The HTML representation of the contact **encodes** the state of the application (that is, exactly what can and cannot be done with this particular representation) in a way that the JSON representation does not.

The client interpreting the JSON response must, again, understand not only the general concept of a Contact, but also specifically what the "status" field with the value "Archived" means. It must know exactly what operations are available on an "Archived" contact, to appropriately display them to an end user. The state of the application, in this situation is not encoded in the response, but rather in a mix of raw data and side channel information such as API documentation.

Furthermore, in the majority of front end SPA frameworks today, this contact information would live *in memory* in a Javascript object representing a model of the contact. The DOM would be updated based on changes to this model, that is, the DOM would "react" to

changes to this backing javascript model (hence the term "reactive" programming, the basis for react and similar SPA frameworks.)

This is certainly *not* using hypermedia as the engine of application state: it is using a javascript model as the engine of application state, and synchronizing that model with a server and with the browser. Let’s call this approach Javascript As The Engine of Application State (JATEOAS).

So, for most javascript applications today, Hypermedia is definitely *not_* the "engine of application state". Rather a collection of javascript model objects living in memory are the engine of application state, with the DOM simply being a display layer being driven by changes to these model objects.

In the HTML approach, the hypermedia is, indeed, the engine of application state: there is no additional model on the client side, and all state is expressed directly in the hypermedia, in this case HTML. As state changes on the server, it is reflected in the representation (that is, HTML) sent back to the client. The client (a browser) doesn’t know anything about Contacts or what the concept of "Archiving" is, or anything else about the domain model for this web application: it simply knows how to render HTML.

By virtue of hypermedia it doesn’t need to know anything about it and, in fact, can react incredibly flexibly to changes from the server because of lack of domain specific knowledge.

HATEOAS & API Churn

Let’s look at a practical example of this flexibility: consider a situation where a new feature is added to our contact application that allows you to send a message to a given Contact. How would this change the two responses from the server?

The HTML representation might now look like this:

```
<html lang="en">
<head>
<h1>Joe Smith</h1>
<div>
    <div>Email: joe@example.bar</div>
    <div>Status: Active</div>
</div>
<p>
    <a href="/contacts/42/archive">Archive</a>
    <a href="/contacts/42/message">Message</a>
</p>
</main>
</body>
</html>
```

The JSON representation might look like this:

```
{
  "name": "Joe Smith",
  "email": "joe@example.org",
  "status": "Active"
}
```

Note that, once again, the JSON representation is unchanged. There is no indication of this new functionality. Instead, a client must **know** about the change, presumably via some shared documentation between the client and the server.

Contrast this with the HTML response. Because of the uniform interface of the REST-ful model and, in particular, because we are using Hypermedia As The Engine of Application State, no such exchange of documentation is necessary! Instead, the client (a browser) simply renders the new HTML with this operation in it, making this operation available for the end user without any additional coding changes.

A pretty neat trick!

Now, in this case, if the JSON client is not properly updated, the error state is relatively benign: a new bit of functionality is simply not made available to users. But let’s consider a more severe change to the API: what if the archive functionality was removed? Or what if the URLs for these operations changed in some way? In this case, the JSON client may be

broken in a much more serious manner.

The HTML response, however, would be simply updated to exclude the removed options or to update the URLs used for them. Clients would see the new HTML, display it properly, and allow users to select whatever the new set of operations happens to be. Once again, the uniform interface of REST has proven to be extremely flexible: despite a potentially radically new layout for our hypermedia API, clients continue to keep working.

Because of this flexibility, hypermedia APIs tend not to cause the versioning headaches that JSON Data APIs do. Once a Hypermedia Driven Application has been "entered" (that is, navigated to through some entry point URL), all functionality and resources are surfaced through self-describing messages. Therefore, there is no need to exchange documentation with clients: the clients simply render the hypermedia (in this case HTML) and everything works out. When a change occurs, there is no need to create a new version of the API: clients simply retrieve updated hypermedia, which encodes the new operations and resources in it, and display it to users to work with.

This is truly some deep magic!

17.2.6. Layered System

The Layered System constraint can be found at https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_6

After the excitement of the uniform interface constraint, the "layered system" constraint is a bit boring, although still very useful: the REST-ful architecture is layered, allowing for multiple servers to act as intermediaries between the client and the eventual "source of truth" server.

These intermediary servers can act as proxies, transform intermediate requests and responses and so forth.

A common modern example if this layering feature of REST is the use of Content Delivery Networks (CDNs) to deliver unchanging static assets to clients more quickly, by storing the response from the origin server in intermediate servers more closely located to the client making a request.

This allows content to be delivered more quickly to the end user and reduces load on the origin server.

17.2.7. An Optional Constraint: Code-On-Demand

The final constraint imposed on a REST-ful system is, somewhat awkwardly, described as an "optional constraint", and can be found here: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_7

In this section, Fielding says:

> REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.
>
> — Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

So, scripting *was* and *is* a native aspect of the original REST-ful model of the web, and, thus something that should be allowed in a Hypermedia Driven Application.

However, in a Hypermedia Driven Application the presence of scripting should *not* change the fundamental networking model: hypermedia should still be the engine of application state and server communication should still consist of hypermedia exchanges rather than, for example, JSON data exchanges.

Today, unfortunately, the scripting layer of the web, that is, JavaScript, is quite often used to *replace* rather than augment the hypermedia model. It is against this trend that this book is written. This does not mean that scripting should not be allowed in a hypermedia application, but rather that it should be done in a certain manner consistent with the REST-ful approach.

17.3. Conclusion

After this deep dive into Chapter 5 of Roy Fielding's dissertation, I hope you have much better understanding of REST, and in particular, the uniform interface and HATEOAS. And I hope you can see *why* these characteristics make hypermedia systems so flexible.

If you didn't really appreciate what REST and HATEOAS meant before now, don't feel bad: it took me over a decade of working in web development, and building a hypermedia-oriented library to boot, to realize just how special HTML and the web is!

[1] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

[2] The counter is a common example widget for UI development tools, a trend that seems to have been started by React. İt's unclear if the "counterexample" pun was intentional.

[3] You can also use a `script` attribute, or `data-script` to please HTML validators.