Hypermedia In Action

1. Reintroducing Hypermedia

This chapter covers

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

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

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

It is sad to say, but in some circles today HTML is viewed almost resentfully: it is considered an awkward, legacy user interface description language that must be used build Javascript-based applications, simply because it 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 ain to show that it is a tremendously innovative, flexible and *simple* way to build robust distributed systems. Not only that, but this hypermedia approach deserves a seat at the table when you, a web developer, are considering the architecture of your next online software system.

1.1. So, What Is Hypermedia?

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

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

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

However, even when working with applications built mainly in HTML, there are nearly always other medias involved: images, videos and so forth. This makes *hypermedia* a more appropriate term 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

Before we discuss the more general concepts of hypermedia, let's take a brief look at a concrete, familiar example of it: 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 don't 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 both HTML, the more you will get out of this book.

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

Here is a simple anchor tag:

Listing 1. 1. A Simple Hyperlink

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

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

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

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

Listing 1. 2. A Simple Form

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

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

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

Here is a visual representation of these two hypermedia interactions:

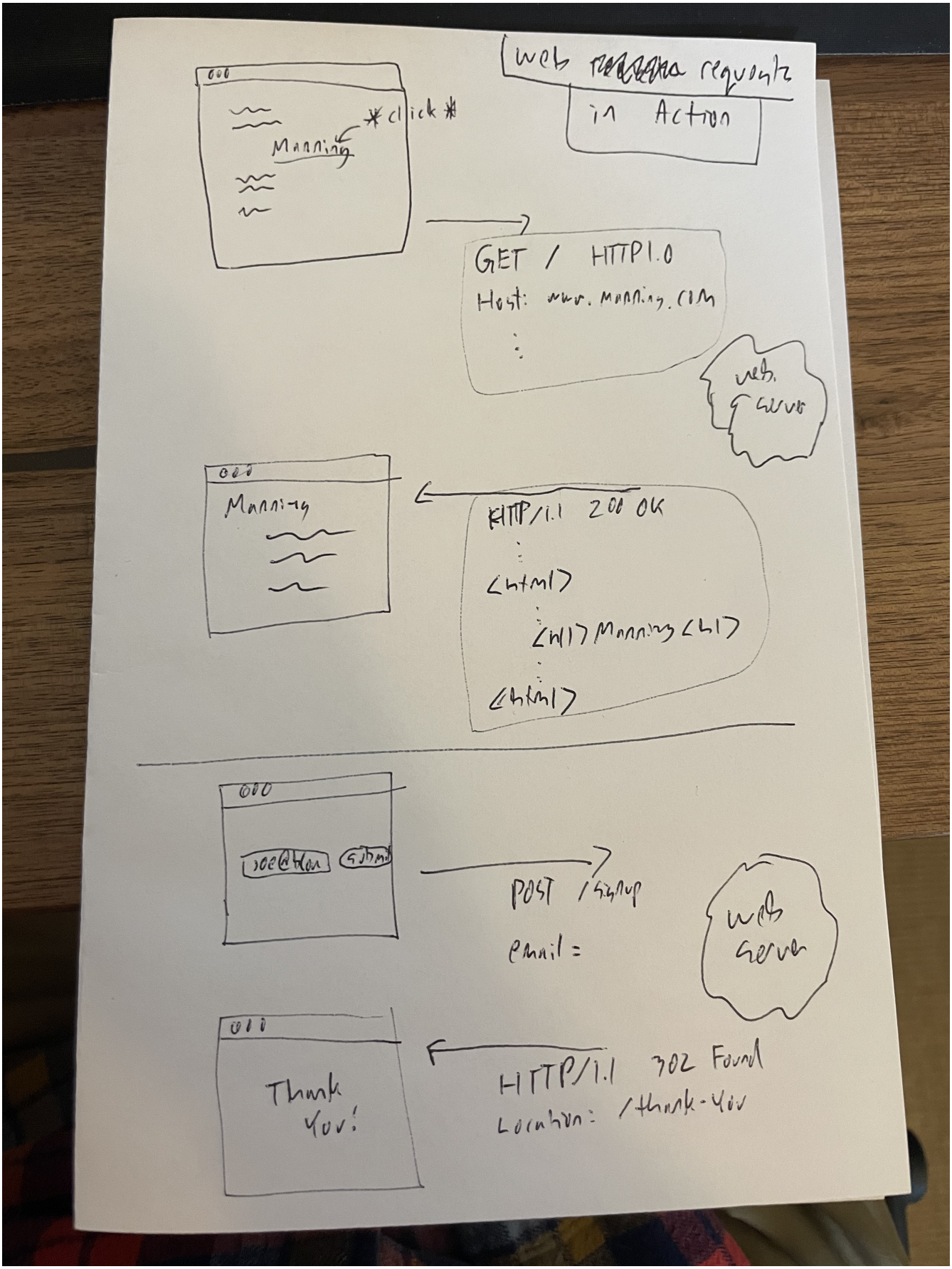

Figure 1. 1. HTTP Requests In Action

As someone interested in web development, the above diagram should look very familiar to you, perhaps even boring. But, despite its familiarity, consider the fact that the two above mechanisms are the *only* easy ways to interact with a server via HTML. That's barely anything at all! And yet,

armed with only these two tools, the early web was able to grow exponentially and offer a staggeringly large amount of 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 perfectly happy with the results.

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 directly. Now let's consider another approach to interacting with a server:

Listing 1. 3. Javascript

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

Here we have a button element in HTML that executes some JavaScript when it is clicked. That JavaScript will issue a `GET` request to `/api/v1/contacts` using the `fetch()` API, a popular API for issuing an "Asynchronous JavaScript and XML", or AJAX request. 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 and it is up to the JavaScript code to deal with the response.

Despite the name, the response to this request will almost certainly be in the JavaScript Object Notation (JSON) format rather than XML. (That is a long story!)

The response to this request might look something like this:

Listing 1. 4. JSON

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

The JavaScript code above converts this JSON text recieved from the server into a Javascript object (which is very easy when using the JSON notation) which is then handed off to the `updateTable()` method. The `updateTable()` method, not shown here to keep things simple, would then update the UI based on the data that has been received from the server.

What we want to stress at this point of the book is that this server interaction is *not* using hypermedia. The JSON API being used here does not return a hypermedia response. This is, rather, a *Data API*, returning simple, plain old domain data in JSON format. It is up to the code in the `updateTable()` method to understand how to turn this plain old data into HTML, which would typically be done via some sort of client-side templating library.

This bit of javascript is the beginnings of what has come to be called a Single Page Application (SPA): the application is no longer navigating between pages using hypermedia. Instead, we are, within a single page, exchanging *data* with the server and updating the content within that page.

Of course, today, the vast majority of web applications adopt far more sophisticated frameworks for managing the user interface than this simple example, libraries like React, Angular, Vue.js, etc. With these more complex frameworks you 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 of your application.) You update these JavaScript objects in memory and then you allow the UI to "react" to those changes via infrastructure baked into the framework itself. (This is where the term "Reactive" programming comes from.)

In this approach, you, the developer, do not interact with hypermedia much at all. You use it to build your user interface, but the anchor tag is de-emphasized and forms become mere data collection mechanisms. Neither interact with the server in their native language of HTML, and instead become user interface elements that drive local interactions with the in memory domain model.

So modern SPAs are much more complex than the basic JavaScript example above. However, at the level of a *network architecture*, these more sophisticated frameworks are essentially equivalent to our simple example: they exchange plain data via JSON with the server rather than exchanging hypermedia.

1.2. Why Use Hypermedia?

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

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

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

Well, it turns out that, even in its original form, the hypermedia architecture has a number of

advantages when compared with the JSON/Data API approach:

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

As someone interested in web development, these advantages no doubt sound appealing to you. The first and last one, in particular, address two pain points in modern web development:

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

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

And it's true: the hypermedia architecture *can* help cure Javascript Fatigue. But you may reasonably be wondering: so, if hypermedia is so great and can address these problems so obvious in the web development industry, why has it has been largely abandoned web developers today? After all, web developers are a pretty smart lot. Why wouldn't they use this obvious, native web technology?

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

This somewhat baffling lack of progress leads immediately to the second and more practical reason that hypermedia has been abandoned: as the interactivity and expressiveness of HTML remained frozen in time, the technology world marched on, demanding more and more interactive web applications. JavaScript, coupled to data-oriented JSON APIs, stepped in as a way to provide these interactive features in web applications to end users. It was this, the *user experience* that really drove the web developer 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 kept pushing it forward and enabling more and more interactivity *within* that original, hypermedia model of the web. If history had worked out that way, perhaps we could have retained much of the simplicity of the original web while still providing better user experiences.

1.2.1. A Hypermedia Comeback?

So, for many developers today working in an industry dominated by JavaScript and SPA frameworks, hypermedia has become a n afterthought, if it is thought of 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.

But, what if history *had* worked out differently?

What if HTML, instead of stalling as a hypermedia, had continued to develop, adding new mechanisms for exchanging hypermedia with servers and increasing its general expressiveness?

What if this made it 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? Could hypermedia be a legitimate architecture to consider when developing a new web application?

The answer is yes and, in fact, in the last decade, some alternative front end libraries (ironically, written in JavaScript!) have arisen that attempt to do exactly this. 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 and, indeed, excellent architectural 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 and thus are, by their nature, hypermedia oriented, if a bit clunky. 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 accepting the less interactive nature inherent to plain HTML.

Some thought leaders in web development, such as Rich Harris, creator of svelte.js, propose a mix of the two styles. Harris calls this approach to building web applications "Transitional", in that it attempts to mix both the old MPA approach and the newer SPA approach in a coherent whole.

Again, 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 feature.

However, by adopting a hypermedia oriented library, it turns out that this interactivity gap closes dramatically between the MPA and SPA approach. A hypermedia oriented library allows you to make the decision based on other considerations, such as overall system complexity.

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

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

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.

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 like this:

Listing 1. 5. an htmx implementation

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

As with the JavaScript example, we 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 response here is expected to be in *HTML format*, not in JSON. This means that htmx is exchanging hypermedia with the server, 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.

So, 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, similarly, the htmx/HDA approach is extremely different from the SPA approach.

This may seem all well and good: a contrived little demo of a simple tool that maybe makes HTML a bit more expressive. But surely this is just a toy! It can't scale up to large, complex modern web applications, can it?

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

1.3. 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 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 fronting a well designed database with simple Create-Read-Update-Delete (CRUD) operations. Again, there is no shame in this!

In any of these later cases, using a hypermedia approach would likely be a great choice: the interactivity needs of these applications are not off the charts, and much of the value lives on the server side, rather on than on the client side. They are all amenable to "large-grain hypermedia data transfers", 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: there is no need for client-side routing, for managing a client side model, for hand-wiring in javascript logic. You will be able to focus your efforts on your server, where your application is actually adding value.

And by layering htmx or another hypermedia-oriented library on top of this approach, you can address many of the usability issues of it with finer-grained hypermedia transfers. This opens up a whole slew of new user interface and experience possibilities, and will be a focus of later chapters.

1.4. When shouldn’t You Use Hypermedia?

But, that being said, there are cases where hypermedia is not the right choice. What would a good example be?

One example that springs 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. Additionally, 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. We would heartily recommend using JavaScript-based infrastructure for building an application like this!

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 high that hypermedia would, in fact, work great for this part of the app.

And, by adopting hypermedia for that part of your application, you could simplify this part of the application quite a bit. You can then save more of your *complexity budget* for the core, complicated spreadsheet logic, keeping the simple stuff simple.

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 that "simplest thing" tool chest.

This brings up two important points:

First, even the most hard core 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.

Second, the hypermedia approach, in both its simple, "vanilla" HTML form and in its more sophisticated htmx 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 favor this approach in general as the ideal way to minimize your web applications complexity.

1.5. Summary

In this chapter, you have been reintroduced the concept of hypermedia and, I hope, you have a better understanding of what it is, and how it is an integral aspect of the web. We also talked about what is *not* hypermedia, focusing on a simple Data-Oriented JSON API. Here are the important points to remember:

- Hypermedia is a unique architecture for building web applications
- Using Data APIs, which is very common in today’s web development world, is very dramatically different than the hypermedia approach
- Hypermedia lost out to SPAs & Data APIs due to interactivity limitations, not due to fundamental

limitations of = the concept

- There is an emerging class of front-end libraries, htmx being one, 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

Before we get into the practical details of implementing a modern Hypermedia Driven Application, in the next chapter we will take a bit of time to make an in-depth study of one of the most influential documents ever written on the hypermedia architecture of the web, the famous Chapter 5 of Roy Fielding's PhD dissertation and the two concepts it defined and codified: REpresentational State Transfer (REST) & Hypermedia As The Engine of Application State (HATEOAS).

2. Hypermedia In Action

3. REST, HATEOAS and All That

This chapter 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)

3.1. Hypermedia, HTML & HTTP: A In-depth Exploration

To reiterate: 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 and the most common hypertext today is the HyperText Markup Language (HTML).

Hyperlinks in HTML are created via anchor tags, and 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
```

And typically consists of at least:

- 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`)

This URL uniquely identifies a retrievable resource on the internet.

A web browser will turn an anchor tag 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, then specifies the path of the resource being requested, finally followed by the HTTP version for this request.

After that are some HTTP *Request Headers*, individual lines of name/value pairs, separated by a colong, 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 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.

Finally, we see some new HTML content, which 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.

HTML also provides form tags for interacting with servers. Form tags can submit either `GET` requests or `POST` requests, discussed in more detail below. Revisiting our simple form from the last chapter, a simple form tag like this:

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

will be rendered as a basic form with a text input and 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) it 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, then specifies the path of the resource being posted to, finally followed by the HTTP version for this request.

Once again we see some request headers, but then 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` does not end up in the browser history. This means that if the user hits the "Refresh" button, the `POST` is not issued. Rather, a `GET` to the final URL is issued. 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 a website is not properly using this Post/Redirect/Get pattern.

3.1.1. 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 it 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 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!

3.2. REpresentational State Transfer (REST)

So, now that we have revisited what hypermedia is and how it is implemented in HTML & HTTP, 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. He 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 is probably 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.

It is important to understand that Fielding considers REST a *network architecture*, that is an entirely different way of architecting a distributed system, when contrasted with earlier distributed systems. REST was and is not simply a checklist 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 check off as you develop a particular system.

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 REST is mainly associated with JSON APIs. I 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, and

discussion how a Data API might be integrated with a Hypermedia Architecture in a later chapter.

But, again: REST describes *the pre-API web*, and letting go of the current

3.2.1. The "Constraints" of REST

Fielding uses various "constraints" to describe how a REST-ful system must behave, giving us an easy way to understand if a system actually satisfies the architectural requirements or not.

- It is a client-server architecture (section 5.1.2) which seems pretty obvious at this point
- 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) which we will discuss below
- 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 in turn.

3.2.2. Client-server Architecture

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, mainly thick-client networking models.

It should be pretty obvious that any web application, regardless of how it is designed, is going to satisfy this requirement.

3.2.3. Statelessness

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 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 up on every request. This session cookie is typically used as a key to look up information stored server side in what is usually termed "the session": things like the current users email or id, their roles, partially created domain objects, catches, and so forth.

This violation of the REST architectural constraints has proven to be useful for web applications and does not appear to have had a significant impact on the overall flexibility of the hypermedia model. It does, however, cause some complexity headaches when deploying hypermedia servers, which, for example, may need to share session state between one another.

3.2.4. Caching

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.

A complete guide to HTTP caching is beyond the scope of this chapter, but will be discussed in more detail later. Suffice to say that HTTP and browser provide this functionality and web applications are able to take advantage of this infrastructure.

3.2.5. The Uniform Interface Constraint

Now we come to the most interesting and, in my 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.

In section 5.1.5 of his dissertation, 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

Let's break down these four additional constraints.

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, solve a 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 that in the `Accept` header in the requests above.)

Self-Descriptive Messages

This constraint (along with the next) form what I consider the crux of the Uniform Interface, of REST and why, in the my opinion, hypermedia is such a powerful network architecture: in a RESTful system, messages must be *self-describing*.

What does that mean?

This means that messages must contain *all information* necessary to both display *and also operate* on the data being represented.

This sounds pretty abstract, so an example will help 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. Feilding noted 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 seen 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 link in it to a page to archive the contact, whereas the JSON representation 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 it is able to update that status, it must know, via some side-channel, exactly how to do so.

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 need only(only!) 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 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 constraint on the Uniform Interface is that, in a REST-ful system, hypermedia should be "the engine of application state".

This 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, once 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 via some other mechanism.

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!

3.2.6. Layered System

After the excitement of the uniform interface constraint, the "layered system" constraint is a bit boring, although still 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.

Again, nothing near as magic as the uniform interface, but still obviously quite useful.

3.2.7. An Optional Constraint: Code-On-Demand

The final constraint imposed on a REST-ful system is, somewhat awkwardly, described as an "optional constraint":

> 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 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 that approach.

We will go into more detail on this matter in the "Scripting In Hypermedia" chapter.

3.3. 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 is!

Of course, traditional Hypermedia Driven Applications were not without issues, which is why Single Page Applications have become so popular. In the next chapter we will introduce a small, simple Contact application written in the old, Web 1.0 style. Then, through the remainder of the book, this application will be updated to demonstrate that it is possible to give it a modern UI, while staying within the hypermedia model and keeping the flexibility and simplicity of that approach.

4. Hypermedia In Action

5. ContactApp

This chapter covers:

- Building a simple contact management web application

- Server Side Rendering (SSR) with HTML

5.1. A Contact Management Web Application

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

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

5.1.1. Which Stack?

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

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

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

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

The HOWL Stack: Hypermedia On Whatever you'd Like

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

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

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

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

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

5.2. Contact.App Functionality

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

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

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

5.3. Our Flask App

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

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

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

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

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

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

5.3.1. Showing A Searchable List Of Contacts

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

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

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

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

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

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

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

{% block content %}

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

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

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

{% endblock %}
```

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

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

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

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

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

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

5.3.2. Adding A New Contact

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

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

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

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

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

{% block content %}

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

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

{% endblock %}
```

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

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

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

associated with the field, if any.

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

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

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

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

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

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

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

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

5.3.3. Viewing The Details Of A Contact

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

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

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

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

Here is what the controller logic looks like:

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

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

The `show.html` template looks like this:

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

{% block content %}

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

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

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


{% endblock %}
```

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

5.3.4. Editing The Details Of A Contact

Editing a contact is definitely more interesting than viewing one.

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

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

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

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

{% block content %}

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

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

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

{% endblock %}
```

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

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

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

Factoring Your Applications

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

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

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

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

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

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

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

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

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

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

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

5.3.5. Deleting A Contact

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

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

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

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

5.3.6. Summary

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

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

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

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

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

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

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