Hypermedia In Action

1. Reintroducing Hypermedia

This chapter covers

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

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

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

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

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

1.1. So, What Is Hypermedia?

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

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

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

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

1.1.1. HTML

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

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

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

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

Here is a simple anchor tag:

Listing 1. 1. A Simple Hyperlink

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

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

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

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

Listing 1. 2. A Simple Form

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

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

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

Here is a visual representation of these two hypermedia interactions:

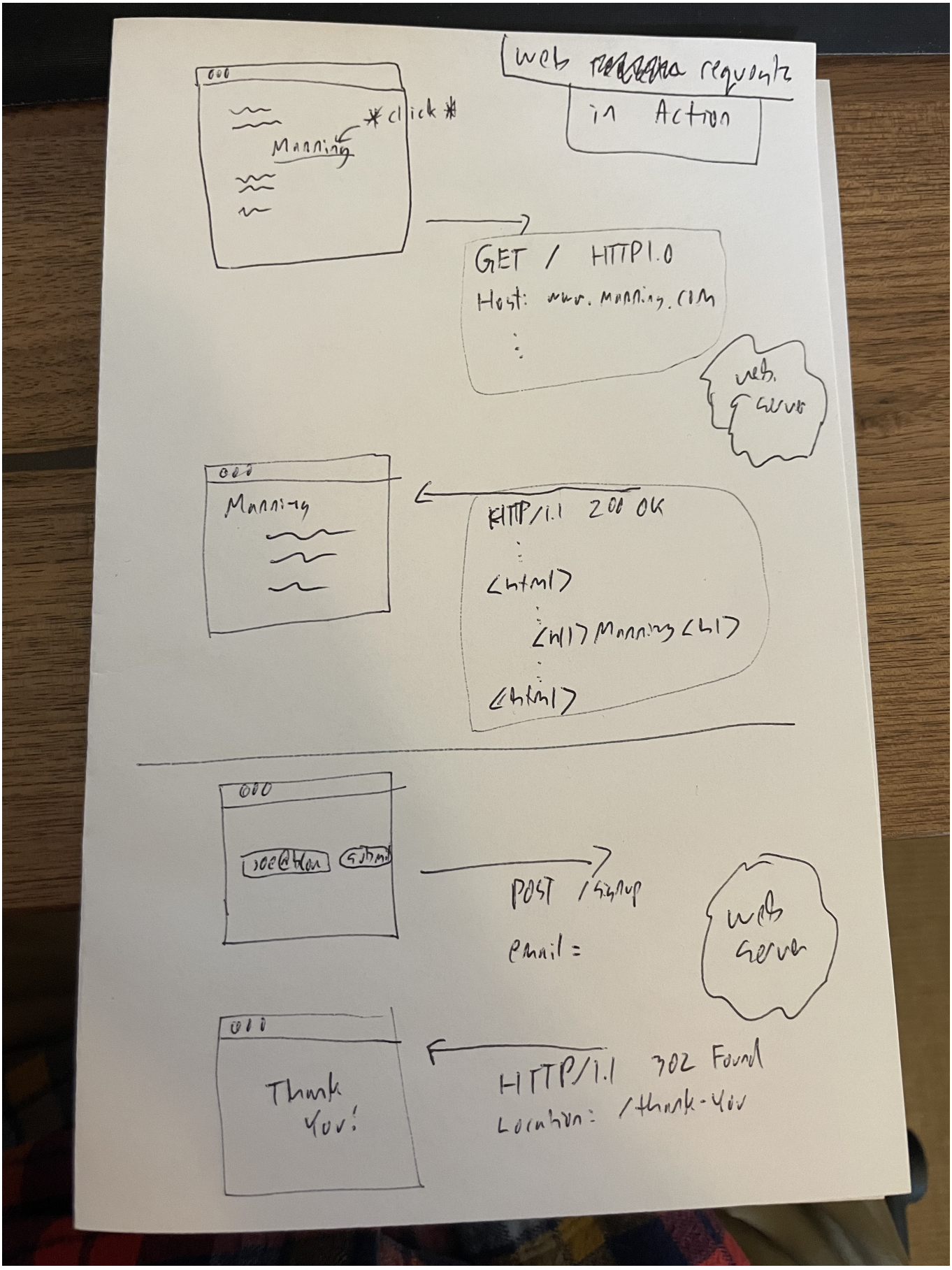

Figure 1. 1. HTTP Requests In Action

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

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

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

1.1.2. So What Isn't Hypermedia?

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

Listing 1. 3. Javascript

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

❶ Issue the request

❷ Convert the response to a JavaScript object

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

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

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

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

Listing 1. 4. JSON

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

❶ The start of a JSON object

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

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

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

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

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

passed into it.

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

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

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

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

1.2. Why Use Hypermedia?

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.1. A Hypermedia Comeback?

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

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

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

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

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

1.2.2. Hypermedia-Oriented Javascript Libraries

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

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

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

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

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

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

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

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

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

It might look something like this:

Listing 1. 5. an htmx implementation

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

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

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

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

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

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

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

experiences.

1.3. REST

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

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

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

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

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

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

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

by the JSON API example and see what happens!

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

1.4. When should You Use Hypermedia?

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

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

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

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

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

experience possibilities. But more on that later.

1.5. When shouldn’t You Use Hypermedia?

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

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

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

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

What Is A Complexity Budget?

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

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

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

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

This brings up two important points:

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

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

1.6. Summary

- Hypermedia is a unique architecture for building web applications
- Using Data APIs, which is very common in today's web development world, is very dramatically different than the hypermedia approach
- Hypermedia lost out to SPAs & Data APIs due to interactivity limitations, not due to fundamental limitations of the concept
- There is an emerging class of Hypermedia Oriented front-end libraries that recenter hypermedia as the core technology for web development and address these interactivity limitations
- These libraries make Hypermedia Driven Applications (HDAs) a more compelling choice for a much larger set of online applications