Hypermedia In Action

1. Hypermedia

This chapter covers

- A reintroduction to the concept 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)* over the *HyperText Transfer Protocol (HTTP)* via a Web Browser on 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 unexplored 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.

It is sad to say, but today HTML is, in some circles, viewed almost resentfully: it is considered a janky, legacy GUI description language that must be used build Javascript-based applications in, simply because that's what happens to be there, in the browser.

This as a shame, and we hope we can convince you that the hypermedia architecture is, instead, a tremendously innovative, flexible and *simple* way to build robust distributed systems. It deserves a seat at the table when you, the developer, are considering the architecture of your next online software system.

1.1. Why Hypermedia?

In this book we aim to re-introduce the reader to the concept of hypermedia, and show that it is a truly unique and powerful *network architecture*, to use the words of Roy Fielding. Fielding was an early developer of web technologies who gave us much of the language we use to discuss the World Wide Web's technical infrastructure.

Fielding recognized that the hypermedia architecture has a number of advantages over other network architectures, as evidenced by the fact that the web grew enormous so quickly: it is extremely simple compared to other approaches to building distributed applications, it survives network outages and changes relatively well and it is extremely tolerant of content and API changes. As someone interested in web development, these features all probably sound pretty appealing to you.

And, I assure you: they are appealing! You may reasonably be wondering: if hypermedia is so great,

why it has been abandoned by so many web developers today. There are, in our opinion, two core reasons: first, hypermedia *as hypermedia* hasn't advanced much since the late 1990s and early 2000s. HTML, the most widely used hypermedia, hasn't added any new ways to interact with a server with pure HTML in nearly two decades.

This somewhat baffling lack of progress has lead to the second reason: JavaScript and data-oriented JSON APIs took over web development as a way to provide more interactive web applications to end users.

It is unfortunate: this did not have to be the case. Rather than abandoning the hypermedia architecture, we could have kept pushing it forward and enabling more and more interactivity *within* that original, hypermedia model of the web. If we had done so, we could have retained much of the simplicity of the original web while still providing better user experiences.

And, in fact, there are some alternative front end libraries today that are attempting to do exactly this. Theese libraries use JavaScript not as a **replacement** for the hypermedia architecture, but rather use it to augment HTML itself *as a hypermedia*. One such library is htmx, was created by the authors and, in later chapters, we will show you how you can build many modern UX patterns for the web using the hypermedia model, often at a fraction of the complexity of more common, JavaScript-oriented solutions.

I this book helps you understand the fundamental REST-ful hypermedia architecture of the original web, and how you might use this architecture to build modern web applications. Even if you choose not to adopt hypermedia as a core technology for your own web development work (and it isn't an appropriate architecture for everything!) then at the very least you should come away with a deeper appreciation for this novel approach to building networked systems and understand where it might be applicable.

1.2. When should You 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 (SPAs) have taken the web development world by storm, offering a far more interactive and immersive experience than any 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 approach for an older, less popular one like hypermedia?

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 what Roy Fielding calls "large-grain hypermedia data transfers".

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.

That does not mean, however, that your hypermedia application has to offer a relatively poor user experience! To the contrary, by using modern hypermedia tools like htmx and with occasional, judicious client side scripting, you can provide excellent user experiences while staying within the simple hypermedia model. For example, you may add an HTML input that filters the results showing in a table *as a user types into the input*, as Google does with search results. Believe it or not, this can be accomplished entirely with hypermedia, no heavy JavaScript required!

I think you will be quite surprised at just how sophisticated our hypermedia-based interfaces can get.

Now, that being said, there are cases where hypermedia is not the right choice. What would an 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. Here we have a highly 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 change would bog performance down terribly. This is simply not a situation amenable to "large-grain hypermedia data transfers" and we would heartly recommend a JavaScript-based approach to an application like this.

However, perhaps this spreadsheet application has a settings page, and perhaps that settings page *is* amenable to a 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 work great for this part of the app. And, by adopting hypermedia for that part of your application, you can save more of your complexity budget for the core, complicate spreadsheet logic.

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.

1.3. OK, 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, making *hypermedia* a more appropriate term for discussing applications built in this manner. We will use the term hypermedia for most of this book, to capture this more general concept.

1.4. 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 get into the more theoretical aspects 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 ninja to understand the code in this book, but the better you understand the core tags and concepts of both HTML and HTTP, the more you will get out of this book.

Now, let's consider the two ur-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' and, when the user clicks on that text, issue an HTTP GET to the url `https://www.manning.com/`. THen take the resulting HTML content from the response and use it to replace the entire screen in the browser."

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 an 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 to the relative URL '/signup'. Take the resulting HTML content in the response 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:

Now, at this point, more experienced developers may be rolling their eyes. "I paid money to read *this*?"

But bear with me!

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 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!

1.5. So What *Isn't* Hypermedia?

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`. The response to this request will be in the JavaScript Object Notation (JSON) format. It is converted to a Javascript object and then handed off to the `updateTable()` method to update the UI based on the data that has been received.

This interaction is *not* using hypermedia. The JSON API being used here does not return a hypermedia response, it is rather a *Data API*, returning simple, plain old domain data. It is up to the browser, in its `updateTable()` method, to understand how to turn this plain old data into HTML, typically via some sort of client-side templating library.

This is a rudimentary single page application: we are not exchanging hypermedia with the server. Instead, we are, within a single page, exchanging *data* with the server and updating that page content.

Of course, today, the vast majority of web applications adopt more sophisticated frameworks for managing the user interface than this simple example: React, Angular, Vue.js, etc. With these more complex frameworks you typically work against a client-side model, updating JavaScript objects in memory and then allowing the UI to "react" to those changes via infrastructure baked into these modern libraries. Hence the term "Reactive" programming.

However, at the level of a network architecture, these more sophisticated are essentially equivalent

to the simple example above: they cast aside the hypermedia network model in favor of a data network model, exchanging JSON with the server.

1.6. Hypermedia Strikes Back

For many developers, since the rise of JavaScript and SPAs, hypermedia has become an 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?

What if 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? Would hypermedia be a legitimate architecture to consider when developing a new web application?

The answer is yes, and there are a few libraries that are attempting to do exactly this: re-center hypermedia as a viable and, indeed, excellent choice for your next web application.

One such library is htmx, which the authors of this book work on, and which will be the focus of much of the remainder of the book. 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 and that, in fact, it is refreshingly simple to do so. And htmx is not alone: other libraries like unpoly.js and hotwire are working in this same conceptual space, making hypermedia, once again, the basis for building web applications.

In the web development world today there is a debate going on between SPAs 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. Many web developers have become exasperated at the complexity of SPA applications and have looked longingly back at the simplicity and flexibility of MPAs.

Some thought leaders in web development, such as Rich Harris, creator of svelte.js, propose a mix of the two styles. Rich 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.

We prefer a slightly different term to MPA. As we wish to emphasize the *hypermedia* aspect of the older (and, with htmx, newer) approach, we like the term *Hypermedia Driven Applications (HDAs)*. This clarifies that the core distinction between this approach and the SPA approach isn't the number of pages in the application, but rather the underlying *network* architecture.

What would the HDA equivalent of the JavaScript-based SPA-style button look like?

Done in htmx, it might look like this:

Listing 1. 4. 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`".

Note especially that the response here is expected to be in HTML format. This means that the htmx interaction is still firmly within the original hypermedia model of the web. Yes, htmx is adding functionality via JavaScript, but that functionality is *augmenting* HTML as a hypermedia, rather than throwing away hypermedia as the network model.

Despite being superficially very similar to one another, it turns out that this example and the JavaScript-based example it is based on are extremely different architectures. And, similarly, this approach is quite different than that taken by most SPA frameworks.

This may seem all well and good: a neat 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 handled internet scale confoundingly well via hypermedia, due to its simplicity this approach 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.

But before we get into the practical details of implementing a modern Hypermedia Driven Application, let's take a bit of time to make an in-depth study of the foundational concepts of hypermedia, and, in particular, of REST & HATEOAS, by reviewing the famous Chapter 5 of Roy Fielding's PhD dissertation on the web.

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 REST-ful 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 tradition. It will not be a great contact management application, because it is going to be designed to be simple and easily demonstrate how to use libraries like htmx, rather than be a real-world, professional application.

None the less, when we are done working with it, we will have some very slick features that many developers would assume would require sophisticated client-side infrastructure. We will implement these features entirely using hypermedia and a bit of light client side scripting.

5.1.1. Which Stack?

For our application we are going to pick a somewhat interesting stack: Python & 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 to execute when a request to that route 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 = Contact.search(search)
    else:
        contacts = Contact.all()
    return render_template("index.html", contacts=contacts)
```

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 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">
        <fieldset>
            <legend>Contact Search</legend>
            <p>
                <label for="search">Search Term</label>
                <input id="search" type="search" name="q" value="{{
request.args.get("q") or '' }}"/>
            </p>
            <p>
                <input type="submit" value="Search"/>
            </p>
        </fieldset>
    </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 be a fairly understandable to 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

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=['POST', 'GET'])
def contacts_new():
    if request.method == 'GET':
        return render_template("new.html", contact=Contact())
    else:
        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)
```

This is a bit more complicated than the `/contacts` handler, but not by a whole lot:

- The `/contacts/new` path is mapped to this python function
    - Note that this route declare that this method should handle both `GET` and `POST` requests made to this path
- If the request is a `GET` we create a new, empty Contact and render the `new.html` template
- If the request is a `POST`, a new contact is created based on the values passed in by a form

- 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.

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.

Once again we are 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, it is simply reflected in the hypermedia response given to users. Users will see the new state of affairs and be able to work with it. No software update required!

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 thelist 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, just look the contact up by id, which is extracted from the end of the path, and display it 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 %}
```

A 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 more interesting than viewing one. Here is the Flask code:

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

As with the `contacts_new` handler, this handler supports both `GET` and `POST`. The logic is very similar to that handler as well:

- Look the contact up by the ID encoded in the path
- If the request is a `GET`, render a form for editing this contact
- If the request is a `POST`, update the contact with the form data submitted
    - If the contact updates successfully, render a flash and redirect

- If not, rerender the `edit.html` form, showing the errors

Once again, Post/Redirect/Get pattern in this control code.

Here is what the `edit.html` template looks like:

```
{% 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 %}
```

Once again, 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 so as to avoid redundancy and only have one place to maintain. Since this is a simple application for demonstrating hypermedia, however, we will keep them separate for now.

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 components that are then composed together on the client side. These components are often developed and tested in isolation and provide a nice abstraction for developers to build with.

In hypermedia applications, in contrast, you factor your application on the server side. The above code could be refactored into a shared template between the two other templates, allowing you to achieve a reusable and DRY (Don't Repeat Yourself) implementation.

Note that the factoring on the server side tends to be coarser-grained than on the client side. This has both benefits (simplicty) and drawbacks (less isolation). 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. (This is a bit junky, more on that in a bit.)

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

5.3.5. Deleting A Contact

The delete functionally only involves a bit of Flask code 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.

No need for a template, the hypermedia response is simply a redirect.

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.