Hypermedia In Action

1. 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)

1.1. 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.

1.1.1. 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!

1.2. 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

1.2.1. 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.

1.2.2. 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.

1.2.3. 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.

1.2.4. 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.

1.2.5. 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!

1.2.6. 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.

1.2.7. 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.

1.3. 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.