Hypermedia In Action

1. Data APIs & Hypermedia Driven Applications

This chapter covers

- Data APIs and how the contrast with hypermedia APIs
- Adding a JSON-Based Data API to our application
- Adding hypermedia controls to our JSON Data API

Data APIs

In this book we have been focusing on using hypermedia to build Hypermedia Driven Applications. In doing so we are following the original networking architecture of the web, building a RESTful system.

However, today, many web applications are not built using this approach. Instead, they use a front end library such as React to interact with JSON API on the server. This JSON API typically does not use hypermedia, but, rather is a *Data API*, that is, it simply returns structured domain data to the client, for the client itself to interpret.

Unfortunately, today, for historical reasons, these JSON APIs are often referred to REST APIs, despite the fact that they are, using the original definition of that term, not actually RESTful.

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

The story of how REST came to mean "JSON APIs" in the industry is a long and sordid long one, and beyond the scope of this book. However, if you are interested, you can refer to an essay entitled "How Did REST Come To Mean The Opposite of REST?" on the htmx website.

In this book we will use the term "Data API" to describe these JSON APIs, while acknowledging that the industry will likely continue to call them "REST APIs" for the foreseeable future.

Now, believe it or not, we *have* been creating a RESTful API for our web application. This may sound confusing to you. API? We have just been creating a web application, with paths that return HTML. How is that a RESTful API?

It turns out that our web application is, indeed, providing an API. It just happens to be an API that a *hypermedia client*, that is, a browser, understands. We are building an API for the browser to interact with over HTTP, and, thanks in no small part to HTML, the hypermedia we are using, we are building a RESTful API. Building web applications like this is so natural and simple that you might not think of it as an API at all, but I assure you, it is!

1.1. Hypermedia APIs & Data APIs

So, we have a hypermedia API for Contact.app. Should we include a Data API as well?

Sure! The existence of a hypermedia API *in no way means* that you can’t also have a Data API. In fact, this is a common situation in traditional web applications: there is a "web app" that is entered through one URL, say `https://mywebapp.example.com/`, and then a separate JSON API that is accessible through another URL, say `https://api.mywebapp.example.com/v1`.

This is a perfectly reasonable way to split up the hypermedia interface to your application and the Data API you provide to other, non-hypermedia clients.

Now, why would you want to include a Data API along with a hypermedia API? Well, because non-hypermedia clients want to interact with your application as well, of course! For example, perhaps you have a mobile application that isn’t built using HyperView (it’s ok, we forgive you). Or maybe you have automated script that needs to interact with the system on a regular basis. Or perhaps there are 3rd party clients who wish to integrate with your system’s data in some way.

For all of these use cases, a JSON Data API makes sense: these are not hypermedia clients so presenting them with a hypermedia API like we have would be inefficient and complicated. A simple JSON Data API fits the bill for what we want and, as always, we recommend using the right tool for the job.

"You Want Me To Parse HTML!?!"

A confusion we often run into in online discussions is that, when we advocate a hypermedia approach to building web applications, people think we mean that they should parse the HTML responses from the server to dump the data into their SPA framework or mobile applications.

This is, of course, silly. What we mean, instead, is that you should consider using a hypermedia API *with a hypermedia client*, like the browser, interpreting the hypermedia response itself and presenting it to the user. If you are writing code to tease apart hypermedia, you are probably doing it wrong.

A lot of confusion around this comes, again, from not understanding that an HTML response from a server *is* and API response, just a very different one than most people think of when writing software.

1.1.1. Differences Between Hypermedia APIs & Data APIs

So, OK, we *are* going to have a Data API for our application. At this point, some developers may be wondering: why have both? Why not have a single API, the JSON Data API, and have multiple clients use this one API to communicate with it? Isn't it redundant to have both types of APIs for our application?

It's a reasonable point: we do advocate having multiple APIs to your web application if necessary and, yes, this may lead to some redundancy in code. However, there are distinct advantages to both sorts of API and, even more so, distinct requirements for both sorts of APIs. By supporting both of these types of APIs separately you can get the strengths of both, while keeping their varying styles of code and infrastructure needs cleanly split out.

Let's consider some of the needs of a JSON Data API:

- It must remain stable over time: you cannot change the API willy nilly or you risk breaking clients that use the API and expect certain end points to behave in certain ways

- It must be versioned: related to the first point, when you do make a major change, you need to version the API so that clients that are using the old API continues to work
- It should be rate limited: since data APIs are often used by other clients, not just your own internal web application, requests should be rate limited, often by user, in order to avoid a single client overloading the system
- It should be a general API: since the API is for *all* clients, not just for your web application, you should avoid specialized end points that are driven by your own application needs. Instead, the API should be general and expressive enough to satisfy as many potential client needs as possible.
- Authentication for these sorts of API is typically token based, which we will discuss in more detail later

Contrast these needs with that of a hypermedia API:

- There is no need to remain stable over time: all URLs are discovered via HTML responses, so you can be much more aggressive in changing the shape of a hypermedia API
- This means that versioning is not an issue, another strength of the hypermedia approach
- Rate limiting probably isn't as important beyond the prevention of Distributed Denial of Service (DDoS) attacks
- The API can be *very specific* to your application needs: since it is designed only for your particular web application, and since the API is discovered through hypermedia, you can add and remove highly tuned end points for specific features or optimization needs in your application
- Authentication is typically managed through a session cookie established by a login page

These two different types of APIs have different strengths and needs, so it makes sense to use both. The hypermedia approach can be used for your web application, allowing you to specialize the API for the "shape" of your application. The Data API approach can be used for other, non-hypermedia clients like mobile, integration partners, etc.

Note in particular that, by splitting these two APIs apart from one another, you reduce the pressure that running your web application through your general Data API produces to be constantly changing the API to address application needs. Rather than being thrashed around with every feature change, your Data API can focus on remaining stable and reliable. This is the core strength of this split API approach, in our opinion.

1.2. Adding a JSON Data API To Contact.app

Alright, so how are we going to add a JSON Data API to our application? One approach, popularized by the Ruby on Rails web framework, is to use the same URL endpoints as your hypermedia application, but use the HTTP `Accept` header to determine if the client wants a JSON representation or an HTML representation. The HTTP `Accept` header allows a client to specify what sort of Multipurpose Internet Mail Extensions (MIME) types, that is file types, it wants back from the server: JSON, HTML, text and so on.

So, if the client wanted a JSON representation of all contacts, they might issue a `GET` request that looks like this:

Listing 8. 1. A Request for a JSON Representation of All Contacts

```
Accept: application/json

GET /contacts
```

If we adopted this pattern then our request handler for `/contacts/ would need to be updated to inspect this header and, depending on the value, return a JSON rather than HTML representation for the contacts. Ruby on Rails has support for this pattern baked into the language, making it very easy to switch on the requested MIME type.

Unfortunately, our experience with this pattern has not been great, for reasons that should be clear given the differences we outlined between Data and hypermedia APIs: they have different needs and often take on very different "shapes", and trying to pound them into the same set of URLs ends up creating a lot of tension in the application code.

So, here, we advocate for applying the Separation of Concerns software design principle and breaking the JSON Data API out to its own set of URLs. This will allow us to evolve the two APIs separately from one another, and give us room to improve each independently in a manner consistent with their own individual strengths.

1.2.1. Picking a Root URL For Our API

Given that we are going to split our JSON Data API routes out from our regular hypermedia routes, where should we place them? One important consideration here is that we want to make sure that we can version our API cleanly in some way, regardless of the pattern we choose. Looking around, a lot of places end up using a sub-domain for their apis, something like `https://api.mywebapp.example.com` and, in fact, often encode versioning in the subdomain: `https://vi.api.mywebapp.example.com`.

While this makes sense for large companies, it seems like a bit of overkill for our modest little Contact.app. Rather than using sub-domains, which are a pain for local development, we will use sub-paths within the existing application:

- We will use `/api` as the root for our Data API functionality
- We will use `/api/v1` as the entry point for version 1 of our Data API

If and when we decide to bump the API version, we can move to `/api/v2` and so on.

This approach isn't perfect, of course, but it will work for our simple application and can be adapted to a subdomain approach or various other methods at a later point, when our Contact.app has taken over the internet and we can afford a large team of API developers. :)

1.2.2. Our First JSON Endpoint: Listing All Contacts

Let's add our first Data API End point. It will handle an HTTP `GET` request to `/api/v1/contacts`, and return a JSON list of all contacts in the system. In some ways it will look quite a bit like our initial code for the hypermedia route `/contacts`: we will load all the contacts from the contacts database and then render some text as a response.

We are going to take advantage of a nice feature of Flask: if you simply return an object from a handler, it will serialized (that is, convert) that object into a JSON response. This makes it very easy to build simple JSON APIs in flask!

Here is our code:

Listing 8. 2. A JSON Data API To Return All Contacts

```
@app.route("/api/v1/contacts", methods=["GET"]) ❶
def json_contacts():
    contacts_set = Contact.all() ❷
    contacts_dicts = [c.__dict__ for c in contacts_set] ❸
    return {"contacts": contacts_dicts} ❹
```

❶ We put our JSON Data API in its own path

❷ We aren't going to support paging or filtering, so we can just load all the contacts here

❸ We convert the contacts array into an array of simple dictionary (map) objects, so they can be serialized to JSON easily

❹ We return a simple dictionary that contains the `contacts` property, pointing to this new array. Flask will automatically serialize this dictionary to JSON for us

The second to last line might look a little funky if you are not a python developer, it is called a "list comprehension", but it's just a way to convert or map a list of values, in this case contacts, to a list of dictionaries or maps. Don't worry about the details, we just want you to understand the general idea: load up all the contacts, do some conversions to make them JSON serializeable, and then return that data structure.

With this in place, if we make an HTTP `GET` request to `/api/v1/contacts`, we will see a response that looks something like this:

Listing 8. 3. Some Sample Data From Our API

```
{
  "contacts": [
    {
      "email": "carson@example.comz",
      "errors": {},
      "first": "Carson",
      "id": 2,
      "last": "Gross",
      "phone": "123-456-7890"
    },
    {
      "email": "joe@example2.com",
      "errors": {},
      "first": "",
      "id": 3,
      "last": "",
      "phone": ""
    },
    ...
   // TODO how to indicate code ommitted
```

So, you can see, a relatively simple JSON representation of our contacts. Not perfect, but good enough for the purposes of this book. This is certainly good enough to, for example, write an automated script against, if, for example, you wanted to move your contacts to another system on a nightly basis.

1.2.3. Adding Contacts

Let's move on the next piece of functionality: adding a new contact to the system. Once again, our code is going to look similar in some ways to the code that we wrote for our normal web application. However, here we are also going to see the JSON API and the hypermedia API for our web application begin to obviously diverge.

In the web application, we needed a separate path, `/contacts/new` to host the HTML form for creating a new contact. In the web application we made the decision to issue a `POST` to that same path to keep things consistent.

In the case of the JSON API, there is no such path needed: the JSON API "just is": it doesn't need to provide any hypermedia representation for creating a new contact. You simply know where to issue a `POST` to to create a contact, likely through some provided documentation about the API, and that's it.

Because of that fact, we can put the "create" handler on the same path as the "list" handler: `/api/v1/contacts`, but have it respond only to HTTP `POST` requests.

The code here is relatively straight forward: populate a new contact with the information from the `POST` request, attempt to save it and, if it is not successful, show some error messages. Here is the code:

Listing 8. 4. Adding Contacts With Our JSON API

```
@app.route("/api/v1/contacts", methods=["POST"]) ❶
def json_contacts_new():
    c = Contact(None, request.form.get('first_name'), request.form.get('last_name'),
request.form.get('phone'),
                request.form.get('email')) ❷
    if c.save(): ❸
        return c.__dict__
    else:
        return {"errors": c.errors}, 400 ❹
```

❶ This handler is on the same path as the first one for our JSON API, but handles `POST` requests

❷ We create a new Contact based on values submitted with the request

❸ We attempt to save the contact and, if successful, render it as a JSON object

❹ If the save is not successful, we render an object showing the errors, with a response code of `400 (Bad Request)`

In some ways similar to our `contacts_new()` handler from our web application (we are creating the contact and attempting to save it) but in other ways very different:

- There is no redirection happening here on a successful creation, because we are not dealing with a hypermedia client like the browser
- In the case of a bad request, we simply return an error response code, `400 (Bad Request)`. This is in contrast with the web application, where we simply re-render the form with error messages in it.

It is these sorts of differences that, over time, build up and make the idea of keeping your JSON and hypermedia APIs on the same set of URLs less and less appealing.

1.2.4. Viewing Contact Details

Next let's make it possible for a JSON API client to download the details for a single client. We will naturally use an HTTP `GET` for this functionality and we will follow the convention we established for our regular web application, and put the path at `/api/v1/contacts/<contact id>`, so, for example, if you want to see the details of the contact with the id `42`, you would issue an HTTP `GET` to `/api/v1/contacts/42`.

This code is quite simple:

Listing 8. 5. Getting the Details of a Contact in JSON

```
@app.route("/api/v1/contacts/<contact_id>", methods=["GET"]) ❶
def json_contacts_view(contact_id=0):
    contact = Contact.find(contact_id) ❷
    return contact.__dict__ ❸
```

❶ Add a new `GET` route at the path we want to use for viewing contact details

❷ Look the contact up via the id passed in through the path

❸ Convert the contact to a dictionary, so it can be rendered as JSON response

Nothing too complicated: we look the contact up by ID, provided in the path to the controller, and look that contact up. We then render it as JSON. You have to appreciate the simplicity of this code!

Next, let's add updating and deleting a contact as well.

1.2.5. Updating & Deleting Contacts

As with the create contact API end point, because there is no HTML UI to produce for them, we can reuse the `/api/v1/contacts/<contact id>` path. We will use the `PUT` HTTP action for updating a contact and the `DELETE` action for deleting one.

Our update code is going to look nearly identical to the create handler, except that, rather than creating a new contact, we will look up the contact by ID and update its fields. In this sense we are just combining the code of the create handler and the detail view handler.

Listing 8. 6. Updating A Contact With Our JSON API

```
@app.route("/api/v1/contacts/<contact_id>", methods=["PUT"]) ❶
def json_contacts_edit(contact_id):
    c = Contact.find(contact_id) ❷
    c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email']) ❸
    if c.save(): ❹
        return c.__dict__
    else:
        return {"errors": c.errors}, 400
```

❶ We handle `PUT` requests to the URL for a given contact

❷ Look the contact up via the id passed in through the path

❸ We update the contact’s data from the values included in the request

❹ From here on the logic is identical to the `json_contacts_create()` handler

Once again, very regular and, thanks to the built-in functionality in Flask, simple to implement.

Let’s look at deleting a contact now. This turns out to be even simpler: as with the update handler we are going to look up the contact by id, and then, well, delete it. At that point we can return a simple JSON object indicating success.

Listing 8. 7. Deleting A Contact With Our JSON API

```
@app.route("/api/v1/contacts/<contact_id>", methods=["DELETE"]) ❶
def json_contacts_delete(contact_id=0):
    contact = Contact.find(contact_id)
    contact.delete() ❷
    return jsonify({"success": True}) ❸
```

❶ We handle `DELETE` requests to the URL for a given contact

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

❸ Return a simple JSON object indicating that the contact was successfully deleted

And, with that, we have our simple little JSON Data API to live alongside our regular web application, nicely separated out from the main web application, so it can evolve separately as needed.

1.2.6. Differences Between Our Hypermedia And JSON APIs

Now, we obviously have a lot more to do if we want to make this a production ready JSON API:

- We don't have any rate limiting, which is important for any publicly facing Data API to avoid abusive clients.
- Even more crucially, there is currently no authentication mechanism. (We don't have one for our web application either!)
- We currently don't support paging of our contact data.
- Lots of small issues that we aren't addressing, such as rendering a proper `404 (Not Found)` response if someone makes a request with a contact id that doesn't exist.

A full discussion around all of these topics is beyond the scope of this book, but I'd like to focus in on one in particular, authentication, in order to show the difference between our hypermedia and JSON API. In order to secure our application we need to add authentication, some mechanism for determining who a request is coming from, and also authorization, determining if they have the right to perform the request.

We will set authorization aside for now and consider only authentication.

Authentication in Web Applications

In the HTML web application world, authentication has traditionally been done via a login page that asks a user for their username (often their email) and a password. This password is then checked against a database of (hashed) passwords to establish that the user is who they say they are. If the password is correct, then a *session cookie* is established, indicating who the user is. This cookie is then sent with every request that the user makes to the web application, allowing the application to know which user is making a given request.

HTTP Cookies

HTTP Cookies are kind of a strange feature of HTTP. In some ways they violate the goal of remaining stateless, a major component of the REST-ful architecture: a server will often use a session cookie as an index into state kept on the server "on the side", such as a cache of the last action performed by the user.

Nonetheless, cookies have proven extremely useful and so people tend not to complain about this aspect of them too much (I’m not sure what our other options would be here!) An interesting example of pragmatism gone (relatively) right in web development.

In comparison with the typical web application approach to authentication, a JSON API will typically use some sort of *token based* authentication: an authentication token will be established via a mechanism like OAuth, and that authentication token will then be passed, often as an HTTP Header, with every request that a client makes.

At a high level this is similar to what happens in normal web application authentication: a token is established somehow and then then token is part of every request. However, in practice, the mechanics tend to be wildly different:

- Cookies are part of the HTTP specification and can be easily *set* by an HTTP Server
- JSON Authentication tokens, in contrast, often require elaborate exchange mechanics like OAuth to be established

These differing mechanics for establishing authentication are yet another good reason for splitting our JSON and hypermedia APIs up.

The "Shape" of Our Two APIs

When we were building out our API, we noted that in many cases the JSON API didn’t require as many end points as our hypermedia API did: we didn’t need a `/contacts/new` handler, for example, to provide a hypermedia representation for creating contacts.

Another aspect of our hypermedia API to consider was the performance improvement we made: we pulled the total contact count out to a separate end point and implemented the "Lazy Load" pattern, to improve the perceived performance of our application.

Now, if we had both our hypermedia and JSON API sharing the same paths, would we want to publish this API as a JSON end point as well?

Maybe, but maybe not. This was a pretty specific need for our web application, and, absent a request from a user of our JSON API, it doesn’t make sense to include it for JSON consumers.

And what if, by some miracle, the performance issues with `Contact.count()` that we were addressing with the Lazy Load pattern goes away? Well, in our Hypermedia Drive Application we can simply revert to the old code and include the count directly in the request to `/contacts`. We can remove the `contacts/count` end point and all the logic associated with it. By the miracle of hypermedia, the system will continue to work just fine!

But what if we had tied our JSON API and hypermedia API together, and published `/contacts/count` as a supported end point for our JSON API? In that case we couldn’t simply remove the end point: a (non-hypermedia) client might be relying on it!

Once again you can see the flexibility of the hypermedia approach and why separating your JSON API out from your hypermedia API lets you take maximum advantage of that flexibility.

1.2.7. The Model View Controller (MVC) Paradigm

One thing you may have noticed about the handlers for our JSON API is that they are relatively simple and regular. Most of the hard work of updating data and so forth is done within the contact model itself: the handlers act as simple connectors that provide a go-between the HTTP requests and the model.

This is the ideal controller of the Model-View-Controller (MVC) paradigm that was so popular in the early web: a controller should be "thin", with the model containing the majority of the logic in the system.

The Model View Controller Pattern

The Model View Controller design pattern is a classic architectural pattern in software development, and was a major influence in early web development. It is no longer emphasized as heavily, as web development has split into front-end and back-end camps, but most web developers are still familiar with the idea.

Traditionally, the MVC pattern mapped into web development like so:

- Model - A collection of "domain" classes that implement all the logic and rules for the particular domain your application is designed for. The model typically provides "resources" that are then presented to clients as HTML "representations".
- View - Typically views would be some sort of client-side templating system, and would render the aforementioned HTML representation for a given Model instance.
- Controller - The controllers job is to take HTTP requests, convert them into sensible requests to the Model and forward those requests on to the appropriate Model objects. It then passes the HTML representation back to the client as an HTTP response.

Thin controllers make it easy to split your JSON and hypermedia APIs out, because all the important logic lives in the domain model that is shared by both. This allows you to evolve both separately, while still keeping logic in sync with one another. With properly built "thin" controllers and "fat" models, keeping two separate APIs both in sync and yet still evolving separately is not as difficult or as crazy as it might sound at first.

1.3. Summary

- Having a Hypermedia Driven API is not mutually exclusive with having a JSON Data API applications