Designing Hypermedia APIs Sample Chapter

Contents

Affordances, or "I don't know what I'm doing"

Why are you reading this book?

I'd hope that you're reading this book because you'd like to build an API of some kind, and are looking for guidance on how. Rephrased in another way, you want to *do* something. This is the same reason that people want to use that API of yours: they want to *do something* with it. There's some sort of task at hand. Some sort of motion. Lots of verbs. In fact, I think verbs are a good way to think about hypermedia APIs: if REST is noun-oriented design, then hypermedia is verb-oriented.

So before we dive into how to go about building your API, I first want to show you a few things about what it means 'to do.' This world view is the kind of mindset that you need to be in to build a hypermedia driven design.

So I'd like to take a look at what it means to be doing some kind of thing. Before that happens, we need to introduce some important vocabulary that we can use to talk about APIs.

Some initial terms

RESTful design is often described as 'resource oriented.' This comes straight from Fielding, actually:

> The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

This brings us to our first bit of terminology, and it's an important one. A **resource** is a name for a conceptual mapping from a name to a set of entities. When I say 'mapping', you can think of a hash map, with arrays in the values:

```
{
  "foo": [1, 2, 3],
  "bar": [3, 4, 5],
}
```

Your API is the hash map itself. A 'resource' is each key-value pair, the mapping from a **name** to a set of **entities**. The names are `foo` and `bar`, and the integers are the **entities**. A **name** is just a label. No funny stuff here! An **entity** is the underlying data that's in your data store. The last bit of terminology here is the **representation**. The representation is the particular format that the set displays itself as.

What it means to be verb oriented

Whew! So what's this have to do with anything? Well, given that the resource enjoys a position as lofty as the 'key abstraction of information,' it only makes sense to feature it, right? This leads to the primary design process being around what resources you have. I've often heard this description of how to design a good API:

> Write down what you want your API to do, and then go through with a highlighter and highlight all the nouns. These nouns will become your resources.

Have you? In a resource-oriented world, you'd want to focus on the resources. It makes perfect sense. But it's the first place in which we'll diverge from what you already know. Resources will still be our primary method of information abstraction, but they won't be our primary method of designing the API.

Why are we making this split? Well, to do this, we need to look to another part of Fielding. One of the ironies of Fielding's thesis is these two sentences:

> 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. These constraints will be discussed in Section 5.2.

If you read section 5.2 in full, you'll discover things about identification of resources. You'll read about manipulating those resources through representations. You'll find out about self-descriptive messages. But you won't find out anything about 'hypermedia as the engine of application state.'

For this, we need to turn to a blog post by Fielding: "REST APIs must be hypertext-driven". In it, Fielding re-emphasises the importance of hypermedia:

> 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?

Yes, there is: 5.2 of the thesis! Luckily, Dr. Fielding expands on what he means:

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

There is a whole lot of greatness packed into this one paragraph, but it gives us the key we need to understand how to design our APIs. That first sentence says it all: almost all of our descriptive effort, ie, our design, is spent in defining the media type used for representing resources and driving application state.

Let's talk about 'driving application state' for a moment. What does 'driving' mean here? This expression calls forth an image of a car. A person who is driving a car is causing the car to be put into motion, and controlling the direction and intensity of that motion. The car is the current state of the application, so to drive that state implies that we want to change it somehow, and we want to be in control of how it changes. We interface with our application through changing the state of these resources, which contain representations of our application's entities.

Repeated a slightly simpler way: our task as an API designer is to figure out a good way to represent the state of our application and then give people tools to modify the state in some way.

The 'resource-first' school of design focuses heavily on the state itself, going so far as to turn the verbs of our API into nouns instead! The issue here is once again well-intentioned, but a mis-reading of the primary information. You see, it *is* true in many ways that the hypermedia style constrains verbs. The misunderstanding here is conflating two different levels of our network stack with each other: the application and the protocol layer.

One of the big seven constraints of REST is the 'uniform interface constraint,' defined in Section 5.1.5 of the dissertation. This section discusses how at the *protocol level*, all APIs must have identical interfaces. This is the reason that HTTP verbs exist, and that there are really only a few of them. Have you ever said something like this: "Ugh, this action doesn't map nicely to GET or POST or PATCH. I wish I could make my own verbs, like COMMIT and LOGIN" ? Fielding knows:

> By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. 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.

Bummer, right? When mapping a service onto a RESTful protocol like HTTP, you won't always get the verbs you want. This has a benefit, though: protocol verbs are global to all applications. If you're building a website built around version control, you might initially add a `COMMIT` verb. But then someone who's writing a database application would get upset, because 'commit' means something very different in their domain. So now we need a process to discriminate and resolve conflict between names. This bureaucracy would slow the rate of change way, way down. This is what Fielding means by 'independent evolveability': my application can change and not affect your application.

However, this is all for the protocol: within the scope of the protocol, we do need to do a variety of things. We can't be super generic, we need specific

names. But we also can't have conflicts. So what do we do?

The answer is that an action in your application is a combination of two things: an action in the protocol, and a name in a namespace that you control. That's it. With APIs on the web, that means "HTTP verb + URL." This is why the documentation around RESTful APIs focuses so much on these two things: "POST /posts : makes a new Post resource." One of the neatest things about URLs that nobody appreciates is that they're basically a global namespace for resolving name conflicts. I can make whatever names I want in my namespace, and you don't need to worry about it.

Anyway, what this all boils down to: users of our API wish to do something with it to accomplish some kind of task. A resource centric design says "Okay: I want to do a thing. First, what are the nouns, and then from there, we'll try to add on some verbs. Possibly by turning them into nouns." Hypermedia-centric design acknowledges that the state transitions are just as important as the state itself, and possibly even more important. Therefore, we say "Okay: I want to do a thing. What is the verb I want, the nouns will fall out of that."

If I was feeling sarcastic, I might try to turn this into VDD: Verb Driven Design! I can just feel the blog posts and conference talks and books.

Ahem.

This is what you need to do: you need to think about doing, don't think about being. The combination of Object Oriented Programming (which often features its own excessive attraction to nouns over verbs) and resource-oriented design have probably led you to have good expertise with 'being driven design.' So let's talk about what it even means to *do things*.

What it means to 'do'

Let's have a little thought experiment here: imagine that it's your birthday. I've given you a gift card to that swanky new resturant downtown, everyone should eat a good meal on their birthday. You're not sure where it is, so you ask, and I tell you, "It's on the corner of 2nd and 3rd." You show up at that corner, but there are four: one resturant on each corner. How do you know which one is which? Well, there's probably a sign outside, so you check it out, and enter the right resturant. You get a table, and since you haven't been here before, you need to know what to eat. So you read the menu, and place your order.

Easy, eh? This is the kind of stuff we do all the time. The stuff we *do*. Note that almost everything in this story is centered around the action that you're taking. There's some description in there too, of course, but our story is driven forward by verbs.

Futhermore, you were in an unfamilliar environment. How'd you know where to go, what to do? Well, you're (probably) an adult, you've been in the world.

You know that things like signs exist, and that you can ask people for directions. In other words, you're aware that your environment will provide certain signals about what you need to do. Often in our lives, these signals are implicit, but sometimes, like when we're driving, they're very explicit. Don't walk. Stop here. Don't go faster than 75. And so on, and on, and on.

Designers and others use the term 'affordance' to describe this phenomena. An affordance is some sort of signal provided by the environment or some object in it. This signal lets you know that you can perform some kind of action with the object or in the environment.

Affordances will be central to our discussion moving forward. If you want to be good at designing hypermedia APIs, I suggest praticing paying attention to various affordances in your natural environment. Give concious thought to the signs you see, the way that you relate to objects in your environment. These interactions can often give clues to useful affordances you can use yourself.

Here are a few of my favorite affordances that you may recognize:

Signs on doors that say 'push' and 'pull.' What I love about these signs is that they're almost universally unnecessary if you get the opener affordance correct. Have you ever seen a sign on a door that says 'pull' and you push anyway, because the door looks like it needs to be pushed? I do, all the time. The handle or bar or knob on the door is its own affordance, and it communicates a certain kind of interaction. The sign is an explicit attempt to override that affordance with a different one. The sign really should say "I know you think this door needs to be pulled, but that's wrong! Please push instead, trust me."

Street signs are an explict affordances, and one of the most pervasive ones that we have in our environments.

Big, giant, red buttons with glass cases over top. These suggest that you can push one and an incredibly powerful state change will occur.

That's just a few. Keep a look out for them! They're everywhere.

Here's another interesting things about affordances: they only indicate the realm of possibility, they don't require actual understanding. For example, imagine that someone is born with a rare disease whereby they cannot understand intutitively how doors work. That person wouldn't die when they saw a door. They could exist in our world, their options are just limited. For a less ridiculous example, when you travel to a foreign country, you won't be able to understand signs that are written in a language that you don't currently speak.

However, that doesn't mean the signs aren't useful. For example, let's say that you're in Tokyo. You want to go visit Ginza, so you ask the internet how to get there. It tells you that you'll want to board the G train at Ueno Station, and get off at Ginza Station. What do you do? Well, you look up translations, right? So you know that Ueno Station will have a sign that says "uenoeki", and that Ginza Station will have the sign "ginzaeki". Even though you don't

understand that “eki” is “station,” you can pattern match the two names and find your way.

This is a pretty accurate description of how a computer interacts with a hypermedia API. It has a list of things that it understands, and even though it doesn't have a deep semantic understanding of what the symbols it is processing mean. This also illustrates that you don't need strong artificial intellegence to navigate a hypermedia API.

It also implies one other thing: if you don't know the meaning of something, you can just ignore it. Something bad might happen to you, but then you can handle that bad event. Not knowing how to proceed in a certain way doesn't prohibit you from proceeding in a totally different way. If you've ever said the words “Oh, so *that* is what that meant!”, you are well aware that full semantic understanding of affordances isn't strictly neccesary to keep the ball rolling. When we get to building clients, this rule will be incredibly important.

The role of affordances in API design

A noun-orented design doesn't allow for affordances because affordances are verbs, not nouns. A good way to think about hypermedia APIs is ‘including affordances to indicate what you can do next.’ In other words, it's using hypermedia as the means of driving application state forward. Is that all coming together for you now?

This is why the only actions a RESTful API can take are the protocol-level ones. They're the only verbs that exist. In order to verb, you need to CRUD up some new noun. There's an old REST joke: the answer to almost every question is “make another resource.” In order to verb, you first must noun-ify it. Here's an example: let's say that we want to process a deposit on a bank account. In a resource-centric design, we'd make two resources: An “account” resource, and a “new deposit” resource. You might imagine documentation like this:

```
GET  /account           provide account details
POST /account/deposit   desposit money into your account
```

Most people would call this a ‘sub-resource,’ but from the orthodox Fielding perspective, the URL details don't matter. It could as easily be

```
GET  /birthday          provide account details
POST /d3adb33f/yup/cool desposit money into your account
```

Doesn't matter. Each one is distinct. Now, as an affordance for humans, I wouldn't suggest #2, ever. But for our purposes, they're two distinct resources.

Anyway, this documentation provides the description of how to actually do the task. Anyone who wanats to know how to do this action needs the documentation in order to know how to accomplish the task. In a hypermedia design, we'd include the affordance inside of the response:

```
{
  "balance": 100,
  "_links": {
   "http://mysite.com/rels/deposit": {"href": "/account/deposit"},
  },
}
```

This is a response using HAL, one hypermedia-enabled media type. HAL includes a `_links` key in with the data to allow for you to include affordances. You don't need the documentation to know that an account allows for deposits, it's right there in the response itself. Now, when I say it's there, I mean that there are two things:

1. Affordances that indicate possible actions.
2. A way to figure out more information about those affordances if you don't know already.

Of course, in order to know that `http://mysite.com/rels/deposit` means "you can deposit money here" takes some degree of human interpretation. I'm not suggesting strong artificial intellegence here. What I am saying is that we can move the possibility of action from the realm of humans reading documentation to the realm of a computer understanding what's up.

What's also interesting is that this affordance comes at little cost. Imagine a client that doesn't understand any affordances; it is strictly pre-programmed to do certain tasks. Like, say, the RESTful API client we mentioned earlier. What kind of information do you think `GET /account` would return?

```
{
  "balance": 100,
}
```

Yup, the same exact thing, just without affordances. This client can still get its job done just fine: it's effectively memorized what tasks it needs to do and in what order. The downside is that if things ever change, a client that understands the concept of affordances can adjust what it's doing to take advantage of these changes. Clients that don't grok affordances could stop working.

Even if a client does understand affordances, that doesn't mean it understands all affordances. For example, the 'full response' of the API above might look more like this:

```
{
  "balance": 100,
  "holder": "Steve Klabnik",
  "_links": {
   "http://mysite.com/rels/deposit": {"href": "/account/deposit"},
    "http://mysite.com/rels/close":   {"href": "/account/close"},
  },
}
```

There's more data and affordances there we didn't even see before! If we were building an application that just let you deposit money into a given account, we might not care about the holder name or the ability to close the account. To a simple client like that, the above response is effectively identical to the one above with only one affordance. It can't properly percieve the extra affordance and data, and that's okay. It knows just enough of the ones to get the job done.

Affordances and Algorithms

Here's another interesting angle on this particular topic: Algorithms are a series of steps. Algorithms are made of verbs, not nouns. Algorithms are a series of state changes in your application's state. A slightly different way of thinking about hypermedia APIs is the usage of affordances to communicate an algorithm to a client.

For example, here's a simple algorithm for ordering a pizza from a website:

1. Get a copy of the menu to see what kinds of toppings you can get.
2. Pick a size of pizza and some toppings, and add them to your order.
3. Once you've added all the pizzas, choose to finalize the transaction.
4. In order to finalize it, your credit card details will be asked for.
5. Wait a while in order to verify that your payment is accepted.
6. A recepit

As an exercise, try to list out the affordances and resources that you'd need to build out this API. Remember that resources are usually nouns or some grouping of nouns, and that affordances are signs of some action that you'd take.

Done? Here's mine:

Affordances: get a menu, add something to an order, finalize the order, supply credit card details, 'please wait,' and get a receipt.

Resources: a menu, a menu item, an order, a receipt.

These are the building blocks that we'd need to build a pizza-ordering algorithm. Any client that can understand these affordances and resources should be able to successfully order a pizza from every environment that has those affordances and resources. Neat, eh?

This is the first step in hypermedia API design, congratulations! The intial act of design involves properly choosing a set of needed affordances and resources. The simplest way to do this is to describe the actual business process you're trying to accomplish, and then paying attention to the nouns and verbs.

Conclusion

In fact, I'm a big fan of the pizza example. It might be because it's a domain that I know well, I worked at a pizza shop for seven years before becoming a professional programmer. And almost everyone has ordered a pizza at some point in their life. For the rest of this book, we'll be using pizza ordering as our motivating example of a hypermedia API. If you hate pizza, it will work well for ordering almost anything. Just substitute some other food or good.

In this chapter, you mastered the concept of affordances, and learned about the differences between resources, their representations, and entities in a system. In the next chapter, we'll get into the nitty-gritty of fully designing and implementing our pizza ordering API. After that, we'll talk about refining it, adding some things, improving our implementation, and discussing common patterns that come up when building APIs in this manner.