apigee

Google Cloud

Web API Design: The Missing Link

Best Practices for Crafting Interfaces that Developers Love

Table of contents

Foreword

The state of the art in web API design is constantly evolving as web APIs continue to become more important in business and in technology.

As a leader in API management, Apigee works with hundreds of customers to develop and manage a large number of APIs. By reflecting on our experiences and those of our customers and the industry at large, we have gained some insights into which API design innovations are bringing real benefits and becoming notable trends.

This book is our attempt to capture some of the significant trends in API design that we have seen emerge in the past couple of years. This book tries to be clear and simple, but it is not intended to be a beginner's guide to API design. If you are looking for more introductory material, you may wish to consult previous books from Apigee on the topic, like this one, or one of many other texts available.

Our earlier book used the example of a simple application for tracking dogs and their owners. In this book, we show how that example might be evolved to match more recent thinking about APIs. Here are two example resources from our earlier book:

```
https://dogtracker.com/dogs/12345678:
{       "id": "12345678",
        "kind": "Dog"
        "name": "Lassie",
        "furColor": "brown",
        "owner": "98765432"
}
```

and

```
https://dogtracker.com/persons/98765432:
{       "id": "98765432",
        "kind": "Person"
        "name": "Joe Carraclough",
        "hairColor": "brown"
}
```

The API allows a client to perform a number of specific operations on these resources. For example, given a dog, retrieve information on the owner, and, given a person, retrieve all the dogs belonging to that person. The common technique for exposing these functions in an API is to define a URI template that the client can use to construct URLs from information in the resources (if you are not familiar with URI templates, they are like URLs with variables in them that can be replaced with values to form a usable URL). In the example above, we might describe the following URI templates:

```
https://dogtracker.com/persons/{personID}
https://dogtracker.com/persons/{personID}/dogs
https://dogtracker.com/dogs/{dogID}
```

These templates allow a client that has access to the data for the dog above to construct the following URL to address the dog's owner:

```
https://dogtracker.com/persons/98765432
```

Similarly, a client can construct this URL to address all the owner's dogs:

```
https://dogtracker.com/persons/98765432/dogs
```

The HTTP methods GET, POST, PUT or PATCH, and DELETE can be used with these templates to read, create, update, and delete description resources for dogs and their owners.

This API style has become popular for many reasons. It is straightforward and intuitive, and learning this pattern is similar to learning a programming language API.

APIs like this one are commonly called RESTful APIs, although they do not display all of the characteristics that define REST (more on REST later).

Introduction

Web APIs use HTTP, by definition. In the early days of web APIs, people spent a lot of time and effort figuring out how to implement the features of previous-generation distributed technologies like CORBA and DCOM on top of HTTP. This led to technologies like SOAP and WSDL. Experience showed that these technologies were more complex, heavyweight, and brittle than was useful for most web APIs. The idea that replaced SOAP and WSDL was that you could use HTTP more directly with much less technology layered on top. Most modern web APIs are much simpler than SOAP or WSDL APIs, but preserve some of the basic ideas of remote procedure call—which is not native to HTTP—implemented much more lightly on top of HTTP. These APIs have come to be known as RESTful APIs. At Apigee, we advocate that you should, as much as you can, only use HTTP without additional concepts.

Here's Why:

When you design any interface, you should try to put yourself in the shoes of the user. As an API provider, you may work on a single API or a small group of APIs, but it is likely that your users deal with many more APIs than yours. This means that they probably come to your API with significant knowledge of basic HTTP technologies and standards, as well as other APIs. Because of this, there is a lot of value in adhering to standards and established conventions, rather than inventing your own. The HTTP specifications are some of the best-written, best-designed, and most universally accepted standards that the industry has ever seen—the chances of you inventing an alternative that will serve your users better are low. Not all your users will have detailed knowledge of the HTTP standards, but making them learn the standard HTTP mechanisms to use your API will be a better investment for them and for you than teaching them an alternative you invented.

For example, if your API uses POST to create a resource, be sure to include a Location header in the response that includes the URL of the newly-created resource, along with a 201 status code—that is part of the HTTP standard. If you need to check that two people don't try to update the same web resource simultaneously, use the ETag and If-Match headers— that again is the HTTP standard. If your API allows users to request data in different formats, use the HTTP Accept header.

If you want to provide alternatives to standard mechanisms that are specific to your API, go ahead, but do it in addition to supporting the standard mechanisms, not instead. Do all of this and your users—especially those that are knowledgeable of the workings of the web and are experienced with other APIs—will thank you.

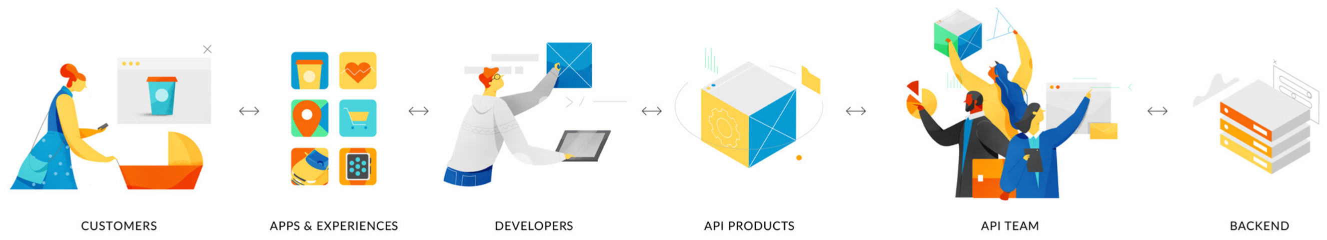
This book is a collection of design practices that we have developed in collaboration with some of the leading API teams around the world. And we'd love your feedback—whether you agree, disagree, or have some additional web API design best practices and tips to share. The API Design group in the Apigee Community is a place where web API design enthusiasts come together to share and debate design practices. We'd love to see you there.

Web APIs and REST

The Job of the API Designer

The API's job is to make the application developer as successful as possible. When crafting APIs, you should think about design choices from the application developer's point of view.

Why? Take a look at the value chain below. The application developer is the linchpin of the entire API strategy. The primary design principle when crafting your API should be to maximize application developer productivity and success.



Getting the design right is important because design communicates how something will be used. The question then becomes— what is the design with optimal benefit for the application developer?

What is a web API?

A web API is the pattern of HTTP requests and responses that is used to access a website that is specialized for access by arbitrary computer programs, rather than (or as well as) web browsers used by humans.

What is REST?

REST is the name that has been given to the architectural style of HTTP itself, described by one of the leading authors of the HTTP specifications. HTTP is the reality—REST is a set of design ideas that shaped it. From a practical point of view, we can focus our attention on HTTP and how we use it to develop APIs. The importance of REST is that it helps us understand how to think about HTTP and its use.

There are a few web APIs that are designed to use only HTTP concepts and no more. The majority of modern APIs uses some subset of the concepts from HTTP, blended with some concepts from other computing technologies. For example, many web APIs are defined in terms of *endpoints* that have *parameters*. *Endpoint* and *parameter* are not terms or concepts that are native to HTTP or REST—they are concepts carried over from Remote Procedure Call (RPC) and related technologies. The term *RESTful* has emerged for web APIs that use more of the native concepts and techniques of HTTP than antecedent technologies did, but also blend in other concepts. Many good APIs have been designed this way, and in fact, this blended style is probably the most common API style in use. HTTP and REST are precisely defined[1], but the term RESTful is not—it is one of those "I know it when I see it" sort of concepts.

We are not purist about HTTP and REST—the effectiveness of the API is what matters—but we also believe that all other things being equal, it is better to add as few concepts beyond just HTTP as possible. If you do this, there will be fewer unique features of your API to be learned and managed over time. This belief is really an expression of the more fundamental idea of Occam's razor—use the fewest number of concepts necessary to solve the problem. Quoting Antoine de Saint-Exupery, "perfection is achieved not when there is nothing more to add, but when there is nothing more to take away."

There are other reasons than conceptual minimalism for limiting ourselves to HTTP concepts for web APIs. One of the most important characteristics that designers of distributed APIs strive for is minimizing coupling between the client and the server. The measure of coupling is how easily either side can be changed without breaking the other. Minimizing coupling is one of the most difficult things to achieve in API design. HTTP's track record of success in this regard may be unprecedented. Consider that the web still supports HTTP clients that were written 20 years ago, showing just how effectively HTTP has decoupled clients from servers. The analysis and description of how HTTP achieves this level of decoupling is one of the highlights of the REST dissertation, and decoupling is one of the primary reasons to be interested in HTTP as a platform for APIs in the first place. One reason that it is controversial to layer concepts from other models on top of HTTP—for example, those that derive from RPC—is that they tend to undermine the loose coupling quality of the HTTP design. This is another important reason to stick with just HTTP.

In the remainder of this book, we explore how to design web APIs using only the inherent concepts of HTTP. We use the term REST API to mean web APIs that have this quality[2].

[1] Sometimes you will hear people claim that REST is not precisely defined—this is not true.

[2] Since REST is an architectural style, it should be possible to invent APIs in this style that are totally independent of HTTP. Such APIs are outside the scope of this book.

HTTP and REST: A Data-oriented Design Paradigm

A REST API focuses on the underlying entities of the problem domain it exposes, rather than a set of functions that manipulate those entities. Following the example introduced in the Foreword, suppose our problem domain is tracking dogs and their owners. Primary entities we might expose would include:

- The collection of known dogs. Its URL might be `https://dogtracker.com/dogs`.
- Individual dogs. They each have a unique URL. We will discuss the format of their URLs in a moment.

We also need something analogous for owners.

Why is a data-oriented approach useful?

Given the URL of an individual dog and knowledge of how the HTTP protocol works, you already know how to do quite a few things. For example, you know how to retrieve the details of the dog using the GET method, delete the dog using the DELETE method, and modify properties of the dog using the PATCH or PUT methods. To use these successfully, you have to understand the properties of a dog, but the mechanics of the operations are already known to you. If the dog has related entities, like an owner or a medical history, you can manipulate those entities in a similar way.

Similarly, given the URL of a collection of dogs, you already know how to create new ones, using the POST method, and find existing ones, using GET. You may have to learn how to filter the collection to narrow your search, but in a well-designed API, it is likely that is done using query parameters added to the URL of the collection.

By contrast, in a function-oriented API, there is more variability, and much more detail you have to learn. Consider this table of examples:

...	...
/getAllDogs	/getAllLeashedDogs
/verifyLocation	/verifyVeterinarianLocation
/feedNeeded	/feedNeededFood
/createRecurringWakeUp	/createRecurringMedication
/giveDirectOrder	/doDirectOwnerDiscipline
/checkHealth	/doExpressCheckupWithVeterinarian
/getRecurringWakeUpSchedule	/getRecurringFeedingSchedule
/getLocation	/getHungerLevel
/getDog	/getSquirrelsChasingPuppies
/newDog	/newDogForOwner
/getNewDogsSince	/getNewDogsAtKennelSince
/getRedDogs	/getRedDogsWithoutSiblings
/getSittingDogs	/getSittingDogsAtPark
/setDogStateTo	/setLeashedDogsStateTo
/replaceSittingDogsWithRunningDogs	
/saveDog	/saveMommaDogsPuppies
...	...

To use an API made up of functions like this, you must understand what a dog is, and what its inherent properties are, but you also have to learn a lot of detail that is specific to the functions of the API. That detail has no clear structure or pattern you can use to help learn it, so there is a significant burden of learning, and the knowledge you acquire will not help you with the next API you need to learn. There are few, if any, common elements between APIs in this style.

A significant part of the value of basing your API design on HTTP and REST comes from the uniformity that it brings to your API. In essence, when you use HTTP natively, you don't need to invent an API at all—HTTP provides the API, and you just define the data in your resources. In REST, this idea is called the uniform interface constraint. In the World Wide Web, the significance of this idea is greater than just making APIs easier to learn—having a uniform interface is part of what makes it possible to implement universal pieces of software, like web browsers and search bots, that work with any website.

API Design Elements

The following aspects of API design are all important, and together they define your API:

- The representations of your resources—this includes the definition of the fields in the resources (assuming your resource representations are structured, rather than streams of bytes or characters), and the links to related resources.
- The use of standard (and occasionally custom) HTTP headers.
- The URLs and URI templates that define the query interface of your API for locating resources based on their data.
- Required behaviors by clients—for example, DNS caching behaviors, retry behaviors, tolerance of fields that were not previously present in resources, and so on.

Designing Representations

Many discussions of API design begin with an extensive discussion of URL design. In a data-oriented model like REST, we think it is better to start with the representation design. We discuss URLs in the section called Designing URLs.

Representation is the technical term for the data that is returned when a web resource is retrieved by a client from a server, or sent from a client to a server. In the REST model, a web resource has underlying state, which cannot be seen directly, and what flows between clients and servers is a representation of that state. Users can view the representation of a resource in different formats, called media types. In principle, all media types for the representation of a particular resource should encode the same information, just in different formats.

Use JSON

The dominant media type for resource representations in web APIs is JavaScript Object Notation (JSON). The primary reasons for JSON's success are probably that it is simple to understand, and it is easy to map to the programming data structures of JavaScript and other popular programming languages (Python, Ruby, Java, and so on). It is now the de facto standard for web APIs, and you should use it.

While JSON is very good and very popular, it is not perfect for our purposes. One limitation is that JSON can only represent a small number of data types (Null, Boolean, Number, String). The most common types we have come across in web API design that are not supported by JSON are dates and times and URLs. There are several options for dealing with this limitation—the simplest is to represent them as strings, and rely on the context to determine which strings are just strings and which are really stringified dates or URLs. We will discuss other options further in the book.

Keep your JSON simple

When JSON is used well, it is simple, intuitive, and largely self-explanatory. If your JSON doesn't look as straightforward as the example below, you may be doing something wrong.

```
{
        “kind”: “Dog”
        “name”: “Lassie”,
        “furColor”: “brown”,
        ...
}
```

The JSON specification says only that a JSON object is a collection of name/value pairs—it does not say what a name is, other than constraining it to be a string. In this example, the names correspond to properties of a web resource whose representation is the enclosing JSON object. These property names are sometimes called predicates, a term borrowed from the theory of grammar. Your JSON will be simpler and easier to understand if you stick to the principle that the names in your JSON are always property names, and the JSON objects always correspond to entities in your API's data model.

Here is an example from the Facebook graph API that shows what JSON looks like when it doesn't follow this advice:

```
{
        "{user-id-a}": {
        "data": [
            {
                "id": "12345",
                "picture": "{photo-url}",
                "created_time": "2014-07-15T15:11:25+0000"
              }
              ... // More photos
           ]
        },
        "{user-id-b}": {
                "data": [
                {
…
```

Notice how user-ids appear on the left of the colon in the position where other examples of JSON we have shown have a property name. The JSON name `data` also does not correspond to a property name in the data model—it is an artifact of the JSON design. We think that JSON like this is more difficult to learn and understand than the simple JSON in our first example and later in examples from Google and GitHub. We recommend that you stick to the simple stuff.

Include Links

Most problem domains include conceptual relationships in addition to simple properties. The natural construct for representing relationships in HTTP is the link.

In years past, the use of links in APIs was strongly associated with the idea that API clients can and should be written to behave like web browsers—meaning that they have no a priori understanding of the data or semantics of any particular API, and their behavior is entirely driven by the contents of the data returned by the server, particularly the links[3]. This idea is sometimes called *Hypermedia As The Engine Of Application State*, or HATEOAS[4] . It makes sense for a web browser to work this way because it would be impractical to write a custom web browser for each site on the web. Clients that work this way are much more difficult and expensive to write than clients that have built-in knowledge of a specific API. In addition to being more expensive to build, generic API clients are usually also less satisfactory, especially clients with a user interface, so this sort of API client is rarely built[5], and the approach is viewed as impractical by most people. A change in recent years has been the realization that the use of links brings a significant improvement to the usability and learnability of all APIs, not just those that are designed to be consumed by completely general-purpose clients.

Reprising the example introduced in the Foreword, assume that there is a relationship between a dog and its owner. A popular way to represent relationship information in JSON looks like the following:

```
{“id”: “12345678”,
“name”: “Lassie”,
“furColor”: “brown”,
“ownerID”: “98765432”
}
```

[3] Of course, JavaScript code that is loaded and executed by browsers often has specific knowledge of a particular API, so even the browser only satisfies the constraints if you exclude the API-specific JavaScript it executes.

[4] At least we think this is what HATEOAS means, and it has certainly been interpreted that way by many. We don’t find the description of HATEOAS in Fielding’s dissertation to be completely clear and unambiguous.

[5] An area where we have seen successful generic API clients driven entirely by data is in the Internet of Things. In the IoT, we have seen a large number of simple device types, each with its own API, where it is cheaper to build a single, data-driven client than to build a custom client for each device.

The ownerID field expresses the relationship. A better way to express relationships is to use links. If your web APIs do not include links today, a first step is simply to add some links without making other changes, like this:

```
{"id": "12345678",
"kind": "Dog"
"name": "Lassie",
"furColor": "brown",
"ownerID": "98765432",
ownerLink": "https://dogtracker.com/persons/98765432"
}
```

If the representation of the owner previously looked like this

```
{"id":"98765432",
"kind": "Person"
"name": "Joe Carraclough",
"hairColor": "brown"
}
```

you can improve it by making it look like this:

```
{"id":"98765432",
"kind": "Person"
"name": "Joe Carraclough",
"hairColor": "brown",
"dogsLink": "https://dogtracker.com/persons/98765432/dogs"
}
```

Why is this better?

In the original example, if you had the representation of Lassie, and you wanted to get the representation of Lassie's owner, Joe, you would have to look in the documentation of the API to find a suitable URI template you could use to construct a URL to locate Joe. In the representation of Lassie, you have an `ownerID`, and you're looking for a URL template that takes an `ownerID` as a variable and produces a URL that locates an owner. This approach has several drawbacks. One is that it requires you to go hunting in the documentation rather than just looking at the data. Another is that there isn't really a good documentation convention for describing which property values found in representations can be plugged into which templates, so some amount of guesswork is usually necessary, even given the right documentation. In this example, you would have to know that owners are people, and look for templates that take the ID of a person as a variable. A third drawback is that you have to write code that will combine the `ownerID` property value with the template and produce a URL. The URI template specification calls such code a template processor. For simple templates, this code is fairly trivial, but it still has to be written and debugged.

By contrast, in our modified design, there is less to learn, and using the API is easier. Joe's URL is right there in the data for you to see, so you don't have to go digging through the documentation for anything. Your client code is also easier to write—simply pick up the value of `ownerLink` and use it in an HTTP request. You don't have to write code to construct a URL from a template and a variable value, and you can even write and use general-purpose code that follows these links without having any built-in knowledge of the API. Adding `dogsLink` to the owner resource provides even more value. Without `dogsLink`, it is not obvious from the data that it is even possible to navigate from an owner to their dogs. An increasing number of APIs from major companies on the web are including links in their data. We'll share some examples from Google and GitHub in the Who uses links? section.

Are URI templates still needed when you have links?

Yes, they are still important. Links express *where you can go from where you are*. They are the signposts for the paved paths to traverse the system. URI templates are like shortcuts, a way to cut across the lawn when you have a very specific destination in mind. In an API that has links, it often happens that there is a handy link in a resource whose representation you already have that takes you where you want to go. It also happens that there is not a link, or that the path to get there by following links is excessively long. In that case, you need another way to locate the resource you need to get to. The URI templates provide a way to do this.

URI templates are most useful if they accept variables whose values are easy for humans to read and remember. For example, the URL template

```
https://dogtracker.com/persons/{personID}
```

has some utility, but remembering the `personID` can be almost as hard as remembering the whole URL if `personID` is some obscure and lengthy identifier. By contrast, the following template

```
https://dogtracker.com/persons/{personName}
```

is more useful because names are very fundamental to the way in which humans deal with the world.

This is the same reason that domain names exist to map names to IP addresses. On the web, when you follow a link to the server that runs the Google Cloud blog, for example, you don't care whether the link contains a domain name or an IP address—you just use it. Having the domain name blog.apgiee.com does improve the experience if you are typing the URL[6].

An analogy with the World Wide Web

The HTML web is a network of documents connected by HTML links. A fundamental way of navigating the web is to follow links to hop from one resource to another. You can also type in a URL directly, or access it from a bookmark, but there is a limit to the number of URLs you can know or bookmark. At the beginning of the web, following links or typing in URLs were the only navigation methods that existed. Some companies (Yahoo! being the most famous) extended the link traversal mechanism to address the problem of discovery of resources. The idea was that https://yahoo.com would be the single root of the web, and from there you could navigate through the links in Yahoo categories to get to anywhere you needed to go. Other companies (Google being the most famous) let you perform a search of web pages using information you know is on those pages (keywords or strings that appeared in the text of the page). Having discovered those pages, you can click on the links they contain to get to new pages until you run out of useful links to click and have to go back to Google to do another search. We now all take it for granted that the way you work with the web is through a combination of search plus link traversal.

[6] Of course, domain names are also important for separating the logical web from the physical infrastructure, even for links.

One way to think about the URI templates of a typical web API is that they are analogous to Google searches. They allow you to locate resources whose URL you don't know using bits of information from those resources that you do know. Most people know how to perform a Google search through a web browser using URLs like this one:

```
https://www.google.com/search?q=web+api+design
```

When web API designers define URL templates that look like this

```
https://dogtracker.com/persons/{personId}/dogs
```

they are in fact specifying a specialized query language for their application. The same query could be expressed as:

```
https://dogtracker.com/search?type=Dog&ownerId={personId}
```

Unlike Google, which provides a universal search language across all websites, URI templates define a query language that is specific to a particular set of resources—your API. Designing a query language specific to your API can be helpful since you can use knowledge of your data and use cases to define a more usable and understandable query capability for your resources and to limit the cost of implementing it. The downside is that the query language for each API has to be learned individually. If there were a universal language for query that was supported by all APIs, the productivity of all API client application developers would be improved. We will look at how to make query URLs regular and predictable in the section entitled Designing query URLs.

Many people have been happy with the success of URI templates for locating resources via queries and have ignored the value of the other mechanism—links. On the other extreme, some link advocates have insisted on a single root model without the use of query/search. These two techniques are valuable complements to each other for web APIs just as they are for the HTML web.

Including links, step 2

If all you do is add URL-valued properties like those shown above, you will already have made a significant improvement to your API. You could make the URL-valued properties read-only, and continue to do updates the way you do today. For example, if you want to change Lassie's owner to the Duke of Rudling, you would modify Lassie using a PUT or PATCH request to change the value of the `ownerID` to be the Duke's ID, and the `ownerLink` property would be recalculated by the server. If you are evolving an existing API, this is a very practical step to take. However, if you are designing a new API, you may want to go one step further with links.

To motivate the next step, consider the following example. Suppose dogs can be owned by either people or institutions (e.g. companies, churches, governments, charities, and NGOs). Our dog tracker site is motivated to allow this model so that they can track ownership of police dogs and other working dogs that do not belong to individuals. Suppose also that the institutions are stored in a different table (or database) from dogs, with their own set of IDs. At this point, it is no longer enough to have an `ownerID` property with a simple value to reference the owner—you also need to specify what type of owner it is, so the server knows what table (or database) to look in. Multiple solutions are possible: You could have `ownerID` and `ownerType` properties, or you could have separate `personOwnerID` and `institutionalOwnerID` properties only one of which may be set at a time. You could also invent a compound `owner` value that encoded both the type and the ID. Each of these solutions may have its merits and demerits, but a very elegant and flexible option is to use links to solve the problem. Imagine that we modify the example to look like this:

```
{     “self”: “https://dogtracker.com/dogs/12345678”,
       “id”: “12345678”,
       “kind”: “Dog”
       “name”: “Lassie”,
       “furColor”: “brown”,
       “owner”: “https://dogtracker.com/persons/98765432”
}
```

The explicit change is that `ownerID` property has been dropped, and there is a single `owner` field that is URL-valued. An implicit change is that the URL-valued `owner` property is now read-write rather than just read-only. You can now easily accommodate both human and institutional owners by simply setting the property to be the URL of a person or the URL of an institution. Clients have to be aware that the `owner` URL may point to different sorts of resources, and they will have to be prepared to react differently according to the data returned when navigating this link. Note that this is exactly the behavior of a web browser—the browser will follow a link in a web page and then act appropriately according to what it finds at the other end (HTML page, XML, PDF, JPEG, and more). This is a powerful and flexible model that we recommend, even though it places some requirements on clients that many are not used to.

A word of caution

We like the model of read-write URL-valued properties a lot, but it has consequences for server implementers. In general, it is not a good idea to store full URLs with domain names in databases. If the domain name of your system ever changes, all those URLs in the databases will be wrong. Sometimes it is unavoidable to change the domain name of a production server, even though this is undesirable. Even if you manage to avoid production server name changes, you want the freedom to use the same database in different environments (like integration test, system test, and pre-production) with different domain names, or just IP addresses. This means that the server should strip the scheme and authority from URLs that identify its own resources before storing them, and put them back when they are requested. Any URLs that identify resources that are completely external to your API will probably have to be stored as absolute URLs. If you are not very experienced with absolute URLs in databases, don't do it.

One way of avoiding the problem of absolute URLs in databases is to accept and produce relative URLs in the API—especially those that start with a single /, called path-absolute URLs in the spec. This is a reasonable approach, and it can avoid a number of implementation issues on the server. The trade-off is that it pushes some burden onto the client because the client will have to turn those relative URLs into absolute ones before it can use them. The good news for clients is that it is always possible to turn a relative URL into an absolute URL using only general knowledge of URL standards—no knowledge specific to your API is needed. The bad news is that in some cases it can be tricky for the client to establish the base to which the URL is relative. If you are willing to learn how to produce and consume absolute URLs on the server, you will create a better API for your users.

In general, writing servers that deal with URLs in resource representations is a skill that needs to be learned. Adding readonly, relative URLs is fairly easy, and if that is the level of your ambition, you can probably just go ahead and do it. Providing absolute URLs, and dealing with read-write URL-valued properties both add some design and implementation problems for servers. The good news is that the solutions to these problems are not very hard, but they are sufficiently subtle that it is easy to get them wrong on the first try.

How should I represent links in my resources?

Our preferred approach for representing links is to use a simple JSON name/value pair, like this:

```
“owner”: “https://dogtracker.com/persons/98765432”
```

What are the advantages to this approach? It is very simple and consistent with the way we represent simple properties in JSON. A disadvantage is that JSON has no encoding for URL values, so you have to encode them as strings. If clients don’t have metadata, they will have to guess which strings are actually URLs, perhaps by parsing all string values or relying on contextual information. This isn’t usually a problem for custom-written client-side code, which generally has contextual information in the sense that it was written by a programmer who had looked at the documentation and knew ahead of time which properties are URL-valued. The problem arises when you’re writing general-purpose clients or library code that can process a representation from any API without knowing its specifics.

There is a significant number of more complex patterns for representing links. We have included some more information on these in the appendix. Whether you like or use any of these or not, we think it is important to note that using links does not require anything more complex than simple JSON properties. We have done a few projects using different styles and we currently favor keeping things as simple as possible.

Who uses links?

The recommendations above for including links in APIs are not the most common practice in published APIs, but they are used in some very prominent ones, such as the Google Drive API and GitHub. Below are examples from the Google Drive API and GitHub. You might think that the problem domain of Google Drive makes it especially suited to these techniques, but we believe that they apply equally to a broad range of problem domains, and we have used them in many of our own designs.

Here's the Google Drive API example:

```
       GET
https://www.googleapis.com/drive/v2/files/0B8G-Akr_SmtmaEJneEZLYjB
         HTTP/1.1 200 OK
         …

         {
              kind": "drive#file",
              "id": "0B8G-Akr_SmtmaEJneEZLYjBBdWxxxxxxxxxxxxxxxx",
              "etag": "\"btSRMRFBFi3NMGgScYWZpc9YNCI/MTQzNjU2NDk3OTU3Nw\"",
              "selfLink": "https://www.googleapis.com/drive/v2/files/0B8G-Akr_Smtm
              "webContentLink": "https://docs.google.com/uc?id=0B8G-Akr_SmtmaEJneE
              "alternateLink": "https://drive.google.com/file/d/0B8G-Akr_SmtmaEJne
              "iconLink": "https://ssl.gstatic.com/docs/doclist/images/icon_12_pdf
              "thumbnailLink": "https://lh4.googleusercontent.com/REcVMLRuNGsohM1C
              "title": "Soils Report.pdf",
              "mimeType": "application/pdf",
         …
              "parents": [
                {
                  "kind": "drive#parentReference",
                  "id": "0AMG-Akr_SmtmUk9PVA",
                  "selfLink": "https://www.googleapis.com/drive/v2/files/0B8G-Akr_Sm
                  "parentLink": "https://www.googleapis.com/drive/v2/files/0AMG-Akr_
                  "isRoot": false
                }
              ],
         …
```

Note: The lines in this example are rather long and are truncated here on the right.

Here is a different example from the GitHub API:

```
{
      “id”: 1,
      “url”: “https://api.github.com/repos/octocat/Hello-World/issues/1347”,
      “repository_url”: “https://api.github.com/repos/octocat/Hello-World”,
      “Labels_url”:”https://api.github.com/repos/octocat/Hello-World/is  sues/1347/labels{/name}”,
      “comments_url”: “https://api.github.com/repos/octocat/Hello-World/issues/1347/comments”,
      “events_url”: “https://api.github.com/repos/octocat/Hello-World/issues/1347/events”,
      “html_url”: “https://github.com/octocat/Hello-World/issues/1347”,
      “number”: 1347,
      “state”: “open”,
      “title”: “Found a bug”,
      “body”: “I’m having a problem with this.”,
      “user”: {
      “login”: “octocat”,
      “id”: 1,
      “avatar_url”: “https://github.com/images/error/octocat_happy.gif”,
      “gravatar_id”: “”,
      “url”: “https://api.github.com/users/octocat”,
      “html_url”: “https://github.com/octocat”,
      “followers_url”: “https://api.github.com/users/octocat/followers”,
      “following_url”: “https://api.github.com/users/octocat/following{/other_user}”,
      “gists_url”: “https://api.github.com/users/octocat/gists{/gist_id}”,
      “starred_url”: “https://api.github.com/users/octocat/starred{/owner}{/repo}”,
      “subscriptions_url”: “https://api.github.com/users/octocat/subscriptions”,
      “organizations_url”: “https://api.github.com/users/octocat/orgs”,
      “repos_url”: “https://api.github.com/users/octocat/repos”,
      “events_url”: “https://api.github.com/users/octocat/events{/privacy}”,
      “received_events_url”: “https://api.github.com/users/octocat/received_events”,
      “type”: “User”, “site_admin”: false
```

```
  },
  “labels”: [
    {
        “url”: “https://api.github.com/repos/octocat/Hello-World/labels/bug”,
        “name”: “bug”,
        “color”: “f29513”
    }
  ],
…
```

You will notice that the GitHub values are not all simple URLs; some of them are URI Templates. Templates allow the URLs of a whole family of resources to be communicated in one string, at the cost of requiring the client to process the template.

More later

There is more to say about designing representations, and we will return to this topic in the section entitled More on representation design. We now turn our attention to URL design.

Designing URLs

In URLs, nouns are good; verbs are bad

A consequence of REST's data-oriented model is that every URL identifies a *thing*. This means that—for well-known URLs and query URLs that are designed to be read and written by programmers of API clients—the URLs should be formed from nouns, which are the natural language part of speech used to designate things.

Well-known URLs

In the example above, the URL for the collection of known dogs is https://dogtracker.com/dogs. In an API, it is always necessary to publish at least one *well-known* URL; otherwise, clients can't get started. We will probably expose at least one more—possibly something like https://dogtracker.com/owners for known owners.

In principle, we could stick with a single well-known URL by publishing a single entity at https://dogtracker.com/ that contained the URLs of all the other useful static entities. It is a good practice to do this because it helps make your entire API discoverable and crawlable. Keep in mind though that it is tedious for clients to have to go to that root every time, so the URLs of all your static entities (ones that came with your application and cannot be deleted) are likely to become well-known to your clients whether you like it or not. Our URL, https://dogtracker.com/dogs, is reasonably memorable, and it is also likely to be a stable URL, unless we shut down our API for some reason, or sell or lose our domain name.

Designing entity URLs

It is not necessary to define any format for the URL of an individual entity for clients, and in general, in a well-crafted web API, clients should not have to construct URLs of individual entities from piece-parts (take a moment to read that sentence again—it is important). When a dog is created, the URL of the new dog will be returned to the client, who may remember it or store it for future reference. Searches made by retrieving [subsets of] https://dogtracker.com/dogs will allow the user to recover forgotten or unknown dog URLs. Therefore, it would be adequate to use dog URLs that look like this:

```
https://dogtracker.com/ZG9n;a8098c1a
```

Computers have no difficulty dealing with URLs that look like this, but humans find them hard, and even though APIs are designed to be used by computers, those computers are programmed by humans. For this reason, it is more common, and usually more desirable, to define entity URLs that are more convenient for people and look like this:

```
https://dogtracker.com/dogs/a8098c1a
```

This URL can still be used as an opaque URL by clients that have no special context, but if a human user already knows that the context is dogs at *dogtracker.com*, then the human only has to remember or recognize **`a8098c1a`**—the rest can be constructed from known context. This is helpful for people, who are good at context but not good at parsing long strings of arbitrary characters. Unfortunately, many APIs always require this kind of contextual knowledge, rather than just accommodating it.

Permalinks

Any link that a server puts in the representation of a resource can be bookmarked, stored by the client, and used later. This means that you need to try to make it stable.

There is some good advice on the web on how to design stable URLs. Here is a good example written by Tim Berners-Lee, which contains the following assertion, "After the creation date, putting any information in the name is asking for trouble one way or another."

A consequence of following this advice is that URLs used in linking tend to be rather unfriendly to humans. Because they must avoid including information that may change, they tend instead to have long strings of random characters in them, like this one from Google:

```
https://docs.google.com/document/d/1YADQAXN2sqyC2OAkblp75GUccnAScVSVI7GZk5M-TRo
```

The popularity of websites like Google Docs and Microsoft OneDrive has perhaps reduced resistance to URLs like these, but it is also important to understand that not all your URLs have to be unfriendly to humans—see the discussion of query URLs later in this document.

From the discussion above, you might think that the ideal format for a URL used in linking would be this:

```
https://dogtracker.com/{uuid}
```

Permalink URLs are designed entirely for the benefit of the server that issues them, not the client that uses them. One common need is to route an incoming GET request to the implementation unit that can process it. Because of this, it is not sufficient for the URL to encode the identity of the resource—the identity of the implementation unit that can handle it has to be there too. Implementation units change over time, so what URL designers need is some identifier that can be flexibly mapped to processing units later. A popular data item to use for this identifier is the principal type of the resource, which is assumed to be immutable. This is why a popular format for permalinks used for linking is:

```
https://dogtracker.com/{type}/{uuid}
```

There is no particular reason to separate the type from the UUID with a slash. It would work equally well to use a different special character, or to use a fixed-length string for the type with no separator at all, but slash is the traditional separator. There are useful pieces of standard implementation software that attach special significance to slashes, so separating the type from the resource ID with a slash is a practical choice.

The web is flat

Hierarchies are everywhere, and hierarchies are an important part of how we think about the world. For example, employees are in departments, which are in business units which are in companies. It is thus very tempting to encode those hierarchies in URLs. For query URLs, this can be a good idea because queries that represent navigation paths through hierarchies (and other relationships) can be very useful for locating things. However, for link URLs that are stored by servers and can be bookmarked by clients, encoding a hierarchy in a URL is a bad idea. Hierarchies are not as stable as they might seem, and encoding them in your URLs will either prevent you from reorganizing your hierarchies or cause your persistent links to break when you do.

Solutions to the renaming dilemma

Here's a situation that many API designers have encountered. You have a resource that has a human-friendly identifier, like a name. It also has, or could have, a machine-generated identifier like a UUID that is pretty unfriendly to humans. You want the URL for the resource to be human-friendly because you understand that URLs are an important part of the API experience. So you face a dilemma:

> If you put the human-friendly identifier in the URL (or otherwise use the human-friendly identifier to persistently reference the resource), then you get human-friendly URLs, but you can never rename the resource because that would break all references. If you put the machine-generated identifier in the URL (or otherwise use the machine-generated identifier to persistently reference the resource), then you can freely rename the resource, but you get URLs that are not friendly to humans.

A solution to this problem is to use one URL in links and offer a different one in URI templates for queries. Take the example of Joe Carraclough shown previously. Joe's name by itself is probably not unique, but we can get Joe to pick a unique one. Let's say he picks `JoeMCarraclough`. This isn't quite as human-friendly as just Joe Carraclough, but it is a lot better than `e9cdcf7a-25b3-11e5-9aec-34363bd0ac10`, which is the sort of thing he'd probably get as a machine-generated ID. We then define a URI template of the following form for queries on persons:

```
https://dogtracker.com/person/{name}
```

As a result of this, we can address Joe with the following URL:

```
https://dogtracker.com/person/JoeMCarraclough
```

This URL is very useful for finding Joe, and it is an attractive URL for inclusion in our API. However, we don't want to use this URL for linking because, while names can be made unique, we need to allow them to change, which is not desirable for persistent links. If we wanted to express a persistent link to Joe, it might look like this:

```
{   "id": "12345678",
   "kind": "Dog"
   "name": "Lassie",
   "furColor": "brown",
   "Owner": "https://dogtracker.com/person/e9cdcf7a-25b3-11e5-34363bd0ac10"
}
```

This isn't a beautiful URL, but that doesn't matter too much because being able to parse URLs in links is not important for users. The URL whose format is more visible to the API user is the query URL derived from the URI template.

When Joe decides to marry his girlfriend and adopt his new wife's last name, Smith, we have no difficulty renaming Joe. All the persistent references to Joe use the machine-generated permalink for Joe, so nothing needs to change there. To find Joe via a query using a URI template, you would provide his new name as follows:

```
https://dogtracker.com/persons/JoeSmith44
```

In the approach above, the following two URLs actually identify different resources:

```
https://dogtracker.com/persons/e9cdcf7a-25b3-11e5-34363bd0ac10 https://dogtracker.
com/persons/JoeMCarraclough
```

The first one identifies a specific person. The second one *identifies whichever person currently has the name JoeMCarraclough*. These two resources may appear to be the same at a particular moment, but may diverge over time. The relationship between these two resources is similar to the relationship between these resources:

```
http://dbpedia.org/page/Donald_Trump
http://dbpedia.org/page/Current_President_Of_the_United_States
```

Although these two URLs[7] actually identify quite different things, they will appear to identify the same thing until the end of Trump's term in office, just as the second link would have previously appeared to identify http://dbpedia.org/page/Barack_Obama until Obama's term in office ended at noon on January 20, 2017.

Another approach we have seen to managing rename is to use aliases. In this approach, the name of the resource is included in the URL, and if the name changes, a new alias URL is added that contains the new name. The same entity can be addressed simultaneously using the old URL with the old name or the new URL with the new name. The approach of introducing aliases can be combined with the use of redirects—typically the older aliases will redirect to the most recent one. Using redirect this way establishes one URL as the canonical URL for the resource, which is often preferable to having multiple peer aliases for the same resource. If you choose the strategy of redirecting to the most recent alias, then the canonical URL changes over time, which can cause problems in some scenarios. GitHub uses aliases with redirects to allow repositories to be renamed. A consequence of this design is that old names are not released for reuse—the URLs corresponding to the old names continue to be associated with the resource forever. Hanging on to old names is not usually a problem, but requires the implementation to store all the previous names for a resource as well as the current one.

[7] The first URL is real; the second we made up.

Stability of types

In the last example, we used `https://dogtracker.com/person/e9cdcf7a-25b3-11e5-34363bd0ac10` as the stable, permanent URL used for linking for Joe. This is based on the assumption that being of type `person` is a stable property. In this example, this is probably OK, but you do need to be thoughtful about types because some types are more stable than others. A person does not change into a dog or a mouse (except perhaps figuratively), but hunters turn into prey, politicians turn into lobbyists, clients are also servers and so on (note that the last example shows that a single resource can have more than one type). The only motivation for having `person` appear in a permalink URL at all is for the server to be able to route incoming requests to the right part of the implementation. So, when you are thinking about this topic, think about it in terms of the mapping—current and future—between resources and implementation handlers, not in terms of the meaning to clients, for whom a link URL should be opaque.

Designing query URLs

Earlier in the book we emphasized *how just using HTTP* makes your API better by reducing the amount that has to be learned that is unique to your API. If you just use HTTP, the only thing that varies from API to API is the data model of the resources. Using HTTP is analogous to using a database management system (DBMS)—once you have learned the API provided by a DBMS, you use that knowledge over and over again for any and every database hosted by that DBMS. All you have to learn when you go to a new database is the schema of the data, not the API for accessing it. The standard API of a DBMS allows you to do basic things like creating, deleting and updating data, but it also allows you to find entities using query. Once you have learned the query language of the DBMS, you already know how to form queries for any database. In the section on designing resources and in the previous section, we discussed the value of having both permalink URLs that are used in links and query URLs that allow resources to be found based on data that is in their state. Query URLs are to HTTP what queries are to a database management system.

When API designers define the URI templates for their API, they are in fact designing a custom query language for the API. Designers have a great deal of freedom in how they define this query language, but the more they exercise this freedom, the greater the burden on the API client to learn the idiosyncrasies of these URI Templates. The goal of this section is to show you how you can design query URLs that are regular and predictable so that a user who knows the model of the data exposed by your API can mostly predict the format of the query URLs without you telling them.

We described earlier the popular pattern of defining query URLs that looks like

```
https://dogtracker.com/persons/{personId}/dogs
```

rather than the following equivalent

```
https://dogtracker.com/search?type=Dog&owner={personId}
```

There are a number of reasons why the former style is usually preferred:

- Many application developers consuming the API find the first style more readable and intuitive.
- API developers implementing the API usually find the first style easier to implement.
- For many people, the first style of query URL does not imply a commitment to a comprehensive query capability, but they are more likely to interpret the second style of query URL to mean that all combinations of query parameters and values are supported.
- This style of query URL can easily express graph traversal queries that are difficult to express in the query parameter style.

An example that illustrates the last point is this:

```
https://dogtracker.com/dogs/123456/owner/spouse
```

The resource identified by this query URL is a resource whose canonical URL might be `https://dogtracker.com/ person/98765`. The query URL encodes a graph query, because it requires first locating the dog whose ID is `123456`, and then navigating the `owner` relationship (a graph edge) to get to the owner, followed by the `spouse` relationship. These sorts of queries are very natural to express in a URL as a sequence of path segments, but much more difficult to express in query parameters.

The rest of this section focuses on the design of query URLs in this style.

Representing relationships in query URLs

Resources almost always have relationships to other resources. In the section entitled Include links, we discussed how to express these relationships in representations. What's a good way to express these relationships in query URLs?

Let's take another look at our `dogtracker` example API. Remember, we had a well-known URL `/dogs` and URLs of the form `/dogs/1234`.

To get all the dogs belonging to a specific owner, we can allow users to do a GET[8] on:

```
/persons/5678/dogs
```

We can define the meaning of this query to be the following:

- Start at the resource whose URL is /.
- Traverse the **`persons`** relationship of the resource at /.
- Select the **`5678`** resource from the multiple values of the **`persons`** relationship.
- Traverse the **`dogs`** relationship of the selected person.

The general pattern of this query URLs is:

```
/{relationship-name}[/{resource-id}]/…/{relationship-name}[/{resource-id}]
```

where the **`{resource-id}`** path segments between square brackets are required only if the relationship in the preceding path segment is multi-valued. This pattern depends on the observation that even the first segment can also be viewed as a relationship name—it is the **`persons`** relationship of the resource whose URL is /.

[8] You could also allow a POST to this URL to create a dog belonging to the person. This would be a shorthand for creating the dog by POSTing to /dogs, and explicitly setting the owner. You can make up your mind on whether or not this shortcut is worth the effort, and whether or not you think it is a good idea to have two ways of doing the same thing.

Express relationships symmetrically in URLs and representations

The query URL above implies that there is a relationship between people and dogs. Your API is easier to learn and understand if you make this relationship explicit in the resource representations too by providing a URL-valued relationship property, as described in the section entitled Include links. The idea is to include the dogs property in every person resource if the dogs relationship is implied by your query URL structure, and, conversely, if a dogs property representing a relationship is present in each person resource, provide a query URL the allows it to be traversed without fetching the person. Our query URLs also imply that / has a persons relationship and a dogs relationship. We can thus infer that its representation should look something like this:

```
{
   “self”: “https://dogtracker.com/”,
   “kind”: “DogTracker”
   “persons”: “https://dogtracker.com/persons”,
   “dogs”: “https://dogtracker.com/dogs”
}
```

A general model for query URLs

Adopting this relationship-traversal interpretation for our query URLs creates a satisfying conceptual unification of the query URLs found in popular web APIs with the single-root with links model of the World Wide Web. The query URLs are simply allowing efficient traversal of sequences of relationships that are already defined in the data without requiring intermediate resources to be fetched. This makes your API easier to learn because it unifies the query URL pattern and the data model. Rapier is an open source project that tries to make it easier to specify web APIs whose query URLs match the underlying data model in this fashion.

There is nothing sacred about this pattern for designing query URLs. Its value is that it makes the query URLs predictable for any client programmer who has already understood the data model of the API. If there is a different pattern for query APIs you like better, use it, but adopting a consistent pattern is almost certainly better than designing each query URL individually. If you find a pattern that you think is superior to ours, share it with us.

Path parameters, or matrix parameters

There is an alternative syntax for the style of query URLs discussed in the previous sections. Rather than / persons/5678/ dogs, we can use URLs like this:

```
/persons;5678/dogs
/persons;id=5678/dogs # synonym for the previous URL
/persons;name=JoeMCarraclough/dogs
```

and generalize them to this rule:

```
        /{relationship-name}[;{selector}]/…/{relationship-name}
[;{selector}]
```

The elements after the semicolon are called path parameters or matrix parameters. We like this style of URL both for its syntactic tidiness and its conceptual clarity because each path segment represents exactly one relationship traversal. In the previous example that did not use path parameters, a path segment would either contain a relationship name or a selector, depending on whether the preceding path segment contained a multi-valued relationship name.

We don't see any downside to path parameters except perhaps that many people are not familiar with them. Jax-RS is the only implementation framework we know that has explicit support for them, although we have implemented them using other frameworks, like Express for Node.js. We imagine that Sinatra, Bottle, or Flask would work too if you are writing in Python or Ruby. Whether or not you use path parameters may ultimately be a matter of individual preference or a practical decision based on implementation considerations.

Filtering collections

The examples above show how effective it is to build the path portion of a query URL using relationship names and selectors that identify a single resource amongst multiple relationship values, but many query APIs have intricacies beyond the traversal of relationships.

It has been traditional to put more complex query clauses that filter collections after the ? in the query string portion of the URL. For example, to get all red dogs running in the park looks like this:

```
/dogs?color=red&state=running&location=park
```

The use of query URLs based on paths and query string parameters can be combined, as shown in this example:

```
/persons/5678/dogs?color=red
```

What about responses that don't involve persistent resources?

API calls that return a response that is not the representation of a persistent resource are common. We've seen it in financial services, Telco, and the automotive domain to some extent.

Words like the following are your clue that you might not be dealing with a persistent resource response:

- Calculate
- Translate
- Convert

For example, say you want to make a simple algorithmic calculation like how much tax someone should pay, or do a natural language translation (one language in the request; another in the response), or convert one currency to another. None involve persistent resources returned from a database.

In these cases, it is not necessary to depart from our noun-based, document-based model. The key insight is that the URL should identify the resource in the response, not the processing *algorithm* that calculates the response. Since the resource whose representation is in the response is a thing, it is easy to invent a URL for it that fits the noun-based entity model of the web. From the client's perspective, it is not relevant whether the result is calculated or retrieved from a database, and indeed it is common for calculated resources to be subsequently stored in a persistent cache for performance.

For example, an API to convert 100 euros to Chinese Yuan:

```
/monetary-amount?quantity=100&unit=EUR&in=CNY
```

or simply

```
/monetary-amount/100/EUR/CNY
```

The resource in the response is an amount of money expressed in a currency, which is a thing you can identify with a noun or noun phrase. You could also implement this example using content negotiation:

Request:

```
GET /monetary-amount/100/EUR HTTP/1.1
Host: currency-converter.com
Accept-Currency:CNY
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 9


683.92CNY
```

In this approach, 100 euros is a monetary amount (a noun), and CNY is the currency unit I'd like to get it in. A third solution for this sort of situation is to POST to some noun-based URL. For example:

Request:

```
POST /currency-converter HTTP/1.1
Host: currency-converter.com
Content-Length: 69
{
:amount”: 100,
“inputCurrency”: “EUR”,
“outputCurrency”: “CNY”
}
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 5



683.92
```

This approach feels very natural to many people, and if this is the best solution for your situation, you can certainly use it. However, it has two downsides you should think about:

- The response is not cacheable using standard HTTP mechanisms. You can make up your own system for caching the response in your application, but to do that you will have to assign a key—an identity—to the response. That key will likely need to include the input amount, the input currency, and the output currency. The key that you are inventing is equivalent to the URL that was defined in the previous solution. If you find yourself defining a cache key like this, or even just thinking that it would make sense to do so, it probably means you should have picked the other design.

- When you use the POST pattern discussed here, you are beginning to stray from the uniform interface model, because each entity you POST to typically has its own unique inputs and outputs that are not deducible from a broader data model that underpins a whole set of operations. This can quickly begin to look more like the wider, more complex interfaces you see in RPC as described in the section Why is a data-oriented approach useful? If this happens, you are beginning to lose one of the most fundamental advantages of REST. Extending the DBMS analogy, this is similar to what happens when you introduce stored procedures into a database schema—it can be powerful, but it makes the interface more complex and increases the coupling between client and server.

You should feel free to use POST in this way if it is the best solution to your problem, but you should not do so if there is a straightforward way to use GET instead, especially if the GET can be designed to address a resource from a data model that underpins other operations of the API.

In summary, there is no reason for a URL to appear to identify a verb—they can always be nouns that identify things.

More on Representation Design

Include self-reference and kind properties

The Google and GitHub examples illustrate two additional design choices that we like. The first is including a `kind` property (sometimes spelled `type`, or `isA`). The second is including a `selfLink` property, which is called url in the GitHub example. Many different spellings of this property are in use—we like the name `self` which is the relationship name standardized in the ATOM specification, registered in IANA, and copied by many others.

Notice particularly that these properties apply to every JSON object, not just the top-level one.

Why are the self and the kind properties good ideas?

Including a `self` property makes it explicit what web resource's properties we are talking about without requiring contextual knowledge—like what URL did you perform a GET on to retrieve this representation—or application-specific knowledge. This is especially important for nested JSON objects where the outer context doesn't help establish what resource we're talking about, but it is good practice everywhere.

Including a `kind` property helps clients recognize whether or not this is an object they know how to process. This helps you add new concepts to your API without having to change the API version.

How should I represent collections?

Collections are very important resources because they give us lists of things, and they are also commonly the things to which we POST to create new resources. What do they look like? As with links, there are more and less complex choices available. First, we will describe our favorite simple solution.

Collections are resources too, so the recommendations we have established thus far for all resources also apply to collections.

For example, use the simplest form of JSON, and give them a **`self`** property and a **`kind`** property. Here is an example of the collection of all dogs:

```
{“self”: “https://dogtracker.com/dogs”,
“kind”: “Collection”,
“contents”: [
       {“self”: “https://dogtracker.com/dogs/12344”,
       “kind”: “Dog”,
       “name”: “Fido”,
       “furColor”: “white”
       },
       {“self”: “https://dogtracker.com/dogs/12345”,
       “kind”: “Dog”,
       “name”: “Rover”,
       “furColor”: “brown”
       }
       ]
}
```

The value of the **`contents`** property is a JSON array of JSON objects. *Note: An array of URLs would be another good option if your clients don’t need any data about each of the content resources.* Each of the nested JSON objects follows the same basic pattern—it has **`self`** and **`kind`** properties in addition to properties that are specific to the domain.

An even simpler option would have been to omit the outer JSON object that defines the collection as a resource, like this:

```
        [
    {"self": "https://dogtracker.com/dogs/12344",
    "kind": "Dog",
    "name": "Fido",
    "furColor": "white"
    },
    {"self": "https://dogtracker.com/dogs/12345",
    "kind": "Dog",
    "name": "Rover",
    "furColor": "brown"
    }
]
```

While we're all for simplicity, we think the extra structure in the first version is worth it. It has the following advantages:

- Collections are like every other resource, not a special case.
- The collection is self-describing, meaning that you don't have to be in the code that just did the GET to know what collection you're looking at.
- There is an obvious place to put other properties of the collection (`creationDate`, `owner`, `modificationDate`, `revisionID`, `parentResource`, and so on).

There is no standard that we know of for the JSON representation for collections illustrated here. This is our best idea on how to represent collections in the spirit of *simple JSON* without inventing a specialized media type, and it has worked well for us in practice.

You sometimes see collection property names customized to the type of its contents by doing something like this:

```
"kind": "Dog-Collection",
"dogs": [ // array of dog JSON objects // ]
...
```

This has the appeal of being descriptive, which is good, but it makes it more difficult to write general-purpose client code that can recognize and process any collection without metadata. Either way is probably fine.

Others have proposed specialized media types for JSON collections, like Collection+JSON. For our own APIs, we think this media type and its protocols introduce too much complexity for the benefit they bring. If you do believe that this media type will work well for your clients and your scenarios, we see no fundamental objection.

Paginated collections

The collection above only had two elements in its contents array, but that particular collection (all dogs) is likely to get large. It is not a good idea for either the client or the server to try to return very large collections, so what we would expect to happen is that the collection would be paginated. Here is an HTTP message exchange that illustrates one way we have handled pagination in the past:

Request:

```
GET /dogs HTTP/1.1
Host: dogtracker.com
Accept: application/json
```

Response:

```
HTTP/1.1 303 See Other
Location: https://dogtracker.com/dogs?limit=25,offset=0
```

Request:

```
GET /dogs?limit=25,offset=0 HTTP/1.1
Host: dogtracker.com
Accept: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Location: https://dogtracker.com/dogs?limit=25,offset=0
Content-Length: 23456

{“self”: “https://dogtracker.com/dogs?limit=25,offset=0”,
“kind”: “Page”,
“pageOf”: “https://dogtracker.com/dogs”,
“next”: “https://dogtracker.com/dogs?limit=25,offset=25”,
“contents”: [
        {“self”: “https://dogtracker.com/dogs/12344”,
        “kind”: “Dog”,
        “name”: “Fido”,
        “furColor”: “white”
        },
        {“self”: “https://dogtracker.com/dogs/12345”,
        “kind”: “Dog”,
        “name”: “Rover”,
        “furColor”: “brown”
        },
        … (23 more)
        ]
}
```

As you can see, the server recognized that returning the whole collection wasn’t going to work, and redirected the client to a different resource that represents the first page of the collection. The client followed the redirect, which returned the first page. The first page contains a subset of the collection contents, references the original collection in the `pageOf` property, and also references the subsequent page in the `next` property (unless there is no next page). There is also a `previous` property that is missing in this case because there is no page before the first one.

IANA keeps a registry of standard link relations that includes `next` and `previous` from the example above. `First` and `last` are also registered with IANA. When you are designing your resource representations, if there is a standard IANA-registered link relation name that fits your meaning, you should use the standard one. Only invent property names for relationships that are specific to your problem domain. If you discover a link relation that is not registered with IANA, and which you think has broader applicability than just your application, you could consider registering it, although registering link relations isn't as easy as registering media types.

> A less popular alternative to IANA link relations is the idea of ontologies that assign URLs to properties. Unless you are working in a community that has already adopted semantic web concepts, we think it is more practical to follow IANA.

There is no standard for the representation of pages of collections shown here—we offer it in the spirit of a minimalist approach that uses JSON in a simple, direct way. There are many other approaches out there.

Custom resource types and using URLs for resource types

To understand our pagination representation, you have to understand that the complete contents of the collection is an array that is specified by the concatenation of the arrays that are the contents of each of the pages. This is not a difficult concept—anyone who has ever read a book is familiar with the idea that the contents of the book are defined by the concatenation of the contents of each page. Nevertheless, this is not a concept inherent in JSON. We didn't declare a custom media type for pagination because these are still just ordinary JSON objects with property, value pairs. It is the meaning of the objects that is specific to us, which is true for all JSON. Confirmation that it is not the media type that is unique comes from the observation that the JSON representation of these objects could be converted to XML, Turtle, and other media types using a standard algorithm that did not require knowledge of collections and pages. If we were going to standardize something, we would standardize the types of objects we are using, not their JSON representation. In other words, rather than standardizing a media type whose name was `vnd.apigee-collections+json`, we would standardize two resource types, `Collection` and `Page` with their properties. These resource types could be represented equally in JSON, HTML, Turtle, XML, and so on. Media types are standardized by registering them in a central place, but if we were standardizing resource types, the web offers a different mechanism, which is to use URLs. We might establish two URLs for our types, like this:

```
https://apigee.com/collections#Collection

https://apigee.com/collections#Page
```

We would then use these URLs directly in our JSON as the value of the `kind` property, instead of the simple strings that are there now. Using URLs for types is a reasonable idea because it avoids conflicts between independent teams whose resources may eventually link to each other, but it does make JSON representations bigger and a bit uglier to read, so it isn't done often. There are currently no standards for our types, and we offer them as examples we have used and like. The pattern is perhaps complex enough that it deserves its own description and if you like this approach, perhaps you would like to collaborate with us on standardization.

Supporting multiple formats

Many modern APIs produce only JSON on output. This has the advantage of simplicity, and you might want to take that path. Depending on your client audience, there may be demand for XML or other formats for your data. If you support multiple output formats, you should do so by supporting the standard HTTP Accept header.

Even if you support only JSON, if you support the PATCH method (you should), you will have to deal with PATCH input formats. The two common input formats for PATCH are JSON Patch and JSON Merge Patch. JSON Merge Patch looks just like an ordinary JSON format, is easier for clients to use, and is easier to implement, but it doesn't support every kind of modification. JSON Patch is more complex to use and implement, but handles more cases. You can implement both—it is not too hard.

The reason you should use PATCH for update is that it helps with the evolution of your API. PUT requires the client to replace the entire content of the resource with every update. As your API evolves, you want to be able to add new properties without breaking existing clients. If clients replace the entire content on every update, you are relying on them to faithfully preserve all properties, even if they weren't changed and even if they didn't exist at the time the client was written. This is fragile—a single misbehaved client can cause significant problems. PATCH avoids this risk. See the section on versioning for more discussion of PATCH.

There is also an argument for producing HTML on output, even for APIs. Many single-page applications (SPAs) that access APIs are completely opaque to search engines. One way to resolve this problem is to use a single set of URLs for the API and SPA and to provide HTML output for each of these URLs. A full discussion of this approach is out of the scope of this book— we'll blog about it.

What about property names?

You have an object with data attributes on it. How should you name the attributes?

Here are API responses from a few leading APIs:

Twitter

```
"created_at": "Thu Nov 03 05:19;38 +0000 2011"
```

Bing

```
"DateTime": "2011-10-29T09:35:00Z"
```

Foursquare

```
"createdAt": 1475795458
```

Arguments over whether snake_case (Twitter) or camelCase (Foursquare) is better have raged for decades (there has even been some scientific research to see which works better). The science is inconclusive, and opinions vary, so pick the one that pleases your customers better. JavaScript, Java, and Objective-C programmers tend to favor camelCase, while Python and Ruby programmers tend to favor snake_case, with PHP being split by sub-communities. You might let the relative popularity of those languages amongst your intended clients be your guide. For most web APIs that means camelCase, so we recommend that you use camelCase unless your intended customer set is atypical.

You should avoid putting characters in property names that are not compatible with identifiers in popular programming languages. A common pitfall is using a hyphen (-), which is common in web standards but is interpreted as the subtraction operator in JavaScript and most other programming languages. There are other characters that are valid in HTTP headers and URLs that should be avoided in property names, like these: `/\=:;,.?#<>@`.

It is also better to avoid using property names that conflict with reserved words and globals in popular programming languages or environments. For example, `self`, `this`, `window`, `document`, `type`, `typeof`, `var`, `const`, `println`, `print`, `console`, `parseInt`, `parseFloat`, `escape`, `unescape`, `function`, `def`, `class`, and so on. Unfortunately, there are lots of these, and it is hard to make a comprehensive list.

Date and time formats

The examples above use different approaches to representing time. There are probably more than a dozen different standards for date and time representation, and the Twitter example doesn't match any that we know of. The format we see used most often is the one standardized by XML Schema, which is a subset of the complex ISO 8601 standard. Bing seems to be using this. Unix had a wonderfully simple and elegant time format that was based on the number of milliseconds elapsed since the beginning of the epoch—defined to be the beginning of 1970, GMT. Foursquare seems to be using this format. This terrifically simple representation—a monotonically-increasing integer that was independent of location and could be differenced to calculate time intervals—made an ideal time representation until it was undermined by a very poor design decision that was made when implementing leap seconds in Unix. The sad story is told here and elsewhere. Even though it is now compromised, this format deserves consideration as a date and time representation.

Chatty APIs

It is a common myth that REST APIs are chatty. If your API is too chatty, it is because you designed the wrong resources for your clients, not because you used REST. The myth of "REST implies chatty" probably comes from the misconception that a REST API must look like a fully normalized relational database schema. In a normalized design, there are separate resources for each distinct conceptual resource, like orders, customers, accounts, and so on. If these are the only resources you define, a UI client that wants to display an order to a user may have to perform many GETs to retrieve the order, its line items, the account, the customer that owns the account, and so on. Fortunately, no rule says that REST interfaces must look like this, any more than there is a rule that a relational database schema must look like this. Normalized schemas make updates simple, so one useful strategy for an API is to define all the resources that you would associate with a normalized schema, and give those resources both read and update capabilities. You can then add as many read-only denormalized resources as you need to support efficient clients. These denormalized resources are true REST resources also—they have meaning as entities, they have URIs, and so on. Their state may overlap with the state of other resources, but this is not a problem, and this API will be no less RESTful and much more useful than a normalized one. Other approaches to denormalization are possible, but many have more complicated implications for update, analogous to the *update through a view problem* of relational databases.

Pagination and partial response

Partial response allows you to give application developers just the information they need.

Take for example a request for a tweet on the Twitter API. You'll get much more than a typical Twitter app often needs— including the name of the person, the text of the tweet, a timestamp, how often the message was retweeted, and a lot of metadata.

Let's look at how several leading APIs handle giving application developers just what they need in responses, including Google who pioneered the idea of partial response.

LinkedIn

```
/people:(id,first-name,last-name,industry)
```

This request on a person returns the ID, first name, last name, and the industry. LinkedIn does partial selection using this terse :(...) syntax which we find elegant, but might be harder for application developers to understand simply because it is less common.

Facebook

```
/joe.smith/friends?fields=id,name,picture
```

Google

```
?fields=title,media:group(media:thumbnail)
```

Google and Facebook have a similar approach, which works well.

They each have an optional parameter called fields after which you put the names of fields you want to be returned.

As you see in this example, you can also put subobjects in responses to pull in other information from additional resources.

Add optional fields in a comma-delimited list

The Google approach works extremely well.

Here's how to get just the information we need from our dogs API using this approach:

```
/dogs?fields=name,color,location
```

This approach works well because:

- It is simple to read.
- A application developer can select just the information an app needs at a given time.
- It cuts down on bandwidth issues, which is important for mobile apps.

The partial selection syntax can also be used to include associated resources cutting down on the number of requests needed to get the required information.

The partial selection syntax can also be used to include associated resources cutting down on the number of requests needed to get the required information.

Make it easy for application developers to paginate objects in a database

In a previous section, we showed how you can use `next`, `previous`, `first`, and `last` links to allow API clients to scroll through large lists very simply without the need to construct URLs from complex URI templates. This is often sufficient for handling collections, but occasionally you may want to allow clients more explicit control over pagination.

Let's look at how Facebook, Twitter, and LinkedIn handle pagination. Facebook uses `offset` and `limit`. Twitter uses `page` and `rpp` (records per page). LinkedIn uses `start` and `count`.

Semantically, Facebook and LinkedIn do the same thing. That is, the LinkedIn `start` and count is used in the same way as the Facebook `offset` and `limit`.

To get records 50 through 75 from each system, you would use:

- Facebook - `offset` 50 and `limit` 25
- Twitter - `page` 3 and `rpp` 25 (records per page)
- LinkedIn - `start` 50 and `count` 25

We like `limit` and `offset`, although the data values that the implementation has to sort on, and the database technologies being used may dictate compromises.

Handling Errors

Most API developers think about the successful paths through the API first and think about exceptions and error handling second, but the quality of error handling is a critical part of the API experience for all clients.

Why is good error design especially important for API designers?

From the perspective of the application developer consuming your web API, everything on the other side of that interface is a black box. Therefore, errors become a key tool providing context and visibility into how to use an API.

First, developers learn to write code through errors. The test-first concepts of the extreme programming model and the more recent test-driven development models represent a body of best practices that have evolved because this is such an important and natural way for developers to work.

Secondly, in addition to when they're developing their applications, developers depend on well-designed errors at the critical times when they are troubleshooting, and resolving issues after the applications they've built using your API are in the hands of their users.

If you have read other parts of this book, it will not surprise you that our advice is to learn the standard HTTP status codes and use them appropriately. Status codes do not exist in isolation—they are part of HTTP response messages that include the status code, headers, and possibly also data. Different status codes come with corresponding response headers in standard patterns that you should know and use. Don't think of status codes as just being return codes from functions you implement. Instead, think of complete HTTP response messages. For example, the `201 Created` status code should always be paired with a `Location` header that provides the URL of the newly-created resource.

```
HTTP/1.1 201 Created
Location: https://dogtracker.com/dogs/1234567
```

If your API allows clients to create resources, this is the standard message pattern for doing it, and you should follow suit rather than making up something of your own. Similarly, the response to a successful GET is not just a 200 status code and some data, it includes a standard package of headers, like this:

```
HTTP/1.1 200 OK
Content-Location: https://dogtracker.com/dogs/1234567
Content-Type: application/json
ETag: 1437080173827
Content-Length: nnnn

// body goes here //
```

Note: `Content-Location` *is an optional header that servers often include only if the 'canonical' URL of the resource is different from the one that was used in the GET request. We think it makes sense to always include this header.*

As a last example, the 405 error code indicating that the client tried to use a method that the server does not support for the given resource must be paired with an Allow header saying which methods are supported for the resource.

```
HTTP/1.1 405 Method Not Allowed
Allow: GET, DELETE, PATCH
```

Learn these combinations of status codes with their matching headers, and the scenarios in which they are used. Whenever you have a similar scenario in your application, make sure you use the same standard pattern. This is part of what it means to be a web API, and it is how you leverage your users' existing knowledge of the web.

Make messages returned in the payload as verbose as possible. Code for code:

`200 – OK`

`401 – Unauthorized`

A message for people

{"developerMessage" : "Verbose, plain language description of the problem for the application developer with hints about how to fix it.", "userMessage":"Pass this message on to the app user if needed.", "errorCode" : 12345, "more info": "https://dev.teachdogrest.com/errors/12345"}

Note: "If the error message is for people, why on earth encode it in JSON", you ask. The answer is that there is usually a programmatic user agent between the user and the message, so the real audience for this message is the user agent.

In summary, be verbose and use plain language descriptions. Add as many hints as your API team can think of about what's causing an error.

We recommend you add a link in your description to more information.

Modeling Actions

Another situation you may encounter is when you want to trigger an action. Let's say you have URLs that identify some process, and you want to be able to start, stop, or pause that process. Many designers resort to a verb-oriented, RPC-like API design in these cases, but there are design options that remain faithful to the noun-based, entity-oriented model of the web. One is to provide a state property of the process and allow clients to set it to started, stopped, or paused. Another option is to allow clients to POST an action request to a related URL. For example, the process resource may have a property called `actionRequests` whose value is a URL, as shown:

```
{"id":"https://example.org/process/123456",

"kind": "Process",

"actionRequests": "https://example.org/processes/123456/requests"

}
```

The client can POST a `StopRequest`, `StartRequest`, `PauseRequest`, or `ResumeRequest` entity to https://example.org/processes/123456/requests. This model is helpful if you want to be able to keep a record of who did what to the process and when, because the POST can create a real persistent entity that holds that information. This model is also helpful when dealing with requests that may take some time to happen, since the POST can return a resource immediately, and the state of that entity can be examined subsequently to determine the progress of the request.

Finally, there is a variant of this design that allows the client to know immediately which requests are valid at a particular point in time. If instead of offering a single `actionRequests` property, we offer separate properties for each request type, we can use the absence or presence of the property to communicate the validity of a particular request at a particular time. If the process has not yet been started, its representation might look like this:

```
{“id”:”https://example.org/process/123456”,
“kind”: “Process”,
“state”: “initial”,
“startRequests”: “https://example.org/process/123456/requests”
}
```

If the process has been started, its representation might look like this:

```
{“id”:”https://example.org/process/123456”,
“kind”: “Process”,
“state”: “running”,
“pauseRequests”: “https://example.org/process/123456/requests”,
“stopRequests”: “https://example.org/process/123456/requests”
}
```

If the process has been paused, its representation might look like this:

```
{“id”:”https://example.org/process/123456”,
“kind”: “Process”,
“state”: “pause”,
“resumeRequests”: “https://example.org/process/123456/requests”,
“stopRequests”: “https://example.org/process/123456/requests”
}
```

It might not matter whether the value of these properties is the same URL or a different one for each. We learned this technique from Siren, which has a more elaborate version of it that you may care to use.

Authentication

Not many years ago, API design guides would contain an involved discussion about which authentication approach to use. At that time the leading websites were using different standards or at different levels. Today, almost every major web API uses OAuth2. Examples are PayPal, Twitter, Google, Facebook, GitHub, and many more. You should do the same.

Using OAuth 2.0 means that web or mobile apps that expose APIs don't have to share passwords. It allows the API provider to revoke tokens for an individual user and for an entire app, without requiring the user to change their original password. This is critical if a mobile device is compromised or if a rogue app is discovered.

Overall, OAuth 2.0 means improved security and better end-user and consumer experiences with web and mobile apps.

Complement with an SDK

If your API follows good design practices, is self-consistent, standards-based, and well documented, and application developers consuming your API may be able to get rolling without a client SDK. Well-documented code samples are also a critical resource. However, most client programmers prefer to use a well-crafted SDK in their implementation language of choice, rather than consume your HTTP API directly. In fact, programmers consuming an SDK will likely think of the language-specific SDK as being the API, ignoring the underlying web API completely. Does this mean your web API design isn't so important after all? Not at all—the quality of the underlying web API has a dramatic impact on the cost and reliability of SDKs and the number of SDKs that get created. The better the web API, the better the SDK, and the greater the likelihood of a quality SDK being developed for a given programming language. The web API will also show through to programmers in various debugging and performance-tuning scenarios. We all turn on the Google Chrome Tools or equivalent to see network messages when we are debugging our Web UIs, right?

Versioning

Versioning of APIs is a controversial topic—you will find a lot of contradictory guidance on the internet. The pattern that is most commonly practiced is probably the one with a version identifier in a path segment of URLs. Since there is little consensus on versioning, simply offering our opinions on the topic may not be very helpful, but we offer two thoughts:

1. Doing nothing at all for API versioning is an intelligent approach that is getting more attention

2. Links and version identifiers in URLs make awkward bedfellows

Doing nothing for versioning

We all understand that APIs need to evolve and that evolving APIs without breaking clients is a challenge, so it may seem surprising that doing nothing in particular for versioning can be an intelligent option. In our APIs, we have noticed that we often started by putting `/v1` as a path segment in URLs, anticipating change, and never changed it. Why is this?

Many changes can be made to an API in a backward-compatible way. If you can make a change in a backward-compatible way without changing version, you should do so. In general, it is safe to add new properties to resources in an API, provided that clients know ahead of time that this may happen. There are a couple of things you can do help with this. One is to help clients test by adding a few random properties to resources on the server. It is not difficult to do, it won't hurt your API, and it allows your clients to be sure that they don't falter when an extra property or two shows up in a resource. Another thing you can do is to use PATCH rather than PUT for updating. The semantics of PUT are that it completely replaces the state of the resource. This means that clients have to include all the properties of any resource they are updating, even properties that didn't exist at the time the client was written. This puts a burden on clients and makes the server itself vulnerable to client errors. PATCH, on the other hand, only changes the data explicitly referenced by the client. The server takes responsibility for the merge, which is safer[9]. Of course, you should never implement a PUT operation that has PATCH semantics—we would hope that goes without saying, although we have seen this done more than once. You can also usually introduce new resource types to your API without breaking existing clients. You have to be a little careful about breaking older clients if references to new types of resources can show up in old data, but you can often avoid this, or train clients to tolerate it.

[9] Perhaps not coincidentally, the SQL UPDATE operation of relational databases has PATCH semantics, not PUT semantics.

From time to time, you will make major fundamental changes to your data model. No amount of versioning cleverness allows a change like this to be made in a way that does not break old clients—you will need to implement a new API and some sort of migration strategy for old data. These kinds of changes are painful, and hopefully infrequent, but they will happen unless your API does not evolve much. Many of the major household-name web companies have gone through at least one transition like this in their APIs.

The idea of including versioning in APIs depends on the assumption that there is a set of changes that are between these extremes—they are too big to be made in a perfectly backward-compatible way, but they are not big enough to require a whole new API. In practice, many people find that this category of change simply doesn't exist for their API—we can say at least that this has happened to us.

You sometimes see it stated that you must include versioning in your API from the beginning because it cannot be added later. In our experience, this is not true. If you release an API without versioning and want to add it later, this is usually easy, regardless of whether you put version identifiers in URLs or headers. Requests that lack a version identifier are considered to be V1 requests. Because it is easy to add later, and because it often turns out not to be necessary, we think that if you are in doubt about what to do about versioning for a new API, you can safely leave it out.

Links and version identifiers in URLs make awkward bedfellows

If you do decide to include versioning in your API, you are likely be faced with the choice of putting a version identifier somewhere in the URL or in a header. If you are also using links, you will find that putting version information in a header is the simpler option. Here is why.

```
GET /v2/dogs/12345678 HTTP/1.1
Host: dogtracker.com
…

HTTP/1.1 200 OK

{ “sel”: “https://dogtracker.com/v2/dogs/12345678”,
     “id”: “1234567”,
     “kind”: “Dog”
     “name”: “Lassie”,
     “furColor”: “brown”,
     “owner”: “https://dogtracker.com/v2/persons/98765432”
}
```

Let’s return to an example we used earlier:

In this example, when the server produced the link https://dogtracker.com/v2/persons/98765432, it was making a guess on which version to include in the URL. Since the client asked for V2 of Lassie the Dog, maybe it is safe to assume that V2 of the person the *Duke of Rudling* is what the client will want? The Duke is a person—do people even have a V2 format? There is also an unpleasant conceptual problem—it is conceptually OK for versions to be resources that have URLs, but Lassie is not owned by a version, she is owned by a person.

One way to resolve both these problems is to invent 1+n URLs—one for the Duke himself and one for each of his versions. The URL of the Duke himself is used in links, and clients can convert that to a version URL if they need to. This solves the problem for the server, which can now always return the URL of the Duke, but it adds a burden to the client to create a version URL from the Duke’s URL. Programmers have to consult out-of-band documentation to learn the pattern for forming version URLs—it cannot be deduced from the data. The API of the OpenStack open source project uses this approach.

One banking organization we know also followed this approach, and made the clients’ job a bit easier by

```
GET /v2/dogs/12345678 HTTP/1.1
Host: dogtracker.com
…

HTTP/1.1 200 OK

{ “self”: “https://dogtracker.com/v2/dogs/12345678”,
     “id”: “12345678”,
     “kind”: “Dog”
     “name”: “Lassie”,
     “furColor”: “brown”,
     “owner”: “/persons/98765432”
}
```

returning relative URLs, as follows:

The URL of the Duke that is returned here is relative—if we resolved it, it would be https://dogtracker.com/persons/ 98765432, since the base to which it is relative is https://dogtracker.com/v2/dogs/12345678. Because the URL was returned in relative form, it is simple for the client to prepend either https://dogtracker.com/v2 or https://dogtracker.com/v1 to get the URL of the version they want. This prepend operation is not a standard URL resolution operation, but it works as a simple string manipulation, which is what most programmers care about. You can make up your mind what the result of a GET on https://dogtracker.com/persons/98765432 should be.

As you can see, putting version identifiers in URLs can be made to work with links, but it entails more conceptual and practical complexity than putting version identifiers in an `Accept-Version` or similar header. Either approach requires custom clients that were programmed using knowledge that had to be communicated in documentation. Of course, not implementing versioning is simpler still, and avoids the need for documentation of special knowledge.

Conclusion

The state of the art in web API design and the pace of innovation have increased in recent years as web APIs become more commercially important.

Web APIs are based on the technologies of the World Wide Web, documented particularly in the HTTP and URI specifications. When designing a Web API, it is best to limit yourself as much as possible to concepts and techniques that are found in those specifications—this minimizes the amount that a client has to learn that is specific to your API. The native concepts of HTTP and URI are entity-oriented, so designing an API in this style is more like designing a database than it is like designing a typical programming-language API. The idea is to focus on your data and just let HTTP provide the uniform API, in the same way that a database management provides a uniform API regardless of the databases that it hosts. Taking an *entity-oriented* approach requires programmers to create a mapping between the API model and their code, which adds some burden. Rewards for this extra effort include the simplicity and uniformity of web APIs, as well as the fact that this approach has proven very successful in maintaining a loose coupling between clients and providers, which is one of the most important and elusive characteristics of a good API.

Because native Web APIs are data-oriented, it is not surprising that a large portion of the effort in designing an API should go into the specification of the formats of the data. HTTP provides a standard API for the basic CRUD[10] portion of every web API, but it does not have as much to say about how queries over the data are expressed. This is why a significant amount of design effort typically goes into the design of URLs that express queries. We have presented some ideas for how you should structure these *query APIs* both to minimize the learning effort for clients and to increase the commonality between different APIs.

Until recently, the use of links in APIs was promoted mainly by a small community that was interested in general-purpose clients that had no a priori knowledge of any particular API, in the same way that a web browser has no a priori knowledge of any particular website. More recently we have seen a growing appreciation for the value of links in APIs for all clients.

Authentication, representations of errors, and SDKs are other topics in API design on which we've tried to give some guidance. Versioning continues to be a topic on which opinions vary, but there has recently been more discussion of the option of doing nothing at all. This option turns out to be quite defensible and works well for many APIs, including some of our own.

[10] Create, Retrieve, Update, Delete.

Resources

Representational State Transfer (REST), Roy Thomas Fielding

2000 RESTful API Design Webinar, 2nd edition, Brian Mulloy, 2011

Apigee Blog

API Craft Google Group

Appendix: Other Approaches to Representing Links

The next-simplest method we know for representing links in JSON looks like this:

```
“owner”: {“href”: “https://dogtracker.com/persons/98765432”}
```

This pattern has the advantage that—provided you know the pattern—you can find all the URL-valued properties without out-of-band information. It also gives a place to put extra link properties. The Terrifically Simple JSON project is one attempt to specify this pattern more completely.

There is another style for representing links in JSON that you will see used.

```
“links”: [
     {“href”: “https://dogtracker.com/persons/98765432”,
     “rel”: “owner”
      }
]
```

Like the previous style, this style has the advantage that you know that the value of the property `owner` is a URL without parsing the value or requiring external metadata (you could do something analogous for dates and other data types too). Disadvantages include the fact that this is more complex, and the name of the property (`owner`) now appears as a value of the `rel` property on the right side of a colon, instead of on the left side where property names normally reside in JSON.

We will let you make up your mind whether or not you like this style. It is standardized (or at least specified) in the JSON hyperschema specification. If you use this style, we think you should consider using a specialized media type, not simply JSON (we wish the JSON hyper-schema specification defined one)—this allows general-purpose clients to interpret your links.

Another format for representing links looks like this:

```
“owner”: {
     “dataType”: “URI”,
     “value”: “https://dogtracker.com/persons/98765432”
     }
```

This format made enough sense to us that we used it on a whole project. You can extend JSON serialization and deserialization on both ends to return a URI object instead of a string, which helps automate the use of this pattern. This approach worked reasonably well, but all of us on the project would regularly forget to do this correctly when we were programming and accidentally use simple strings for URLs, which caused bugs. Ultimately, we decided to stop fighting our own programming instincts and go back to using simple strings. If you use this format, you should again consider using a specialized media type. RDF/JSON uses a version of this technique, even for base types that JSON handles natively.

There are several other specifications to choose from that define more complex approaches to link representation, like Siren, HAL, JSON-LD, RDF/JSON, Collection+JSON, Hydra, and probably several more. Each of these adds significant features and concepts beyond just how to represent links.

Now that you've finished reading,
why stop learning?

MORE ABOUT WEB API DESIGN

Take a deeper dive and explore more videos, eBooks, and articles on Web API Design.

VISIT HERE

VIRTUAL API JAM

Join us for a hands-on workshop that will jumpstart your API program. Our experts will walk you through five lab exercises that will help you design, secure, and analyze APIs that developers love.

RESERVE YOUR SEAT

FREE TRIAL

Explore Apigee Edge, a full lifecycle API management platform that helps you manage the entire API lifecycle from design through iteration—so you can get your API products into the hands of developers today.

TRY IT FREE

apigee

Share this eBook

on social

with a colleague

Google Cloud