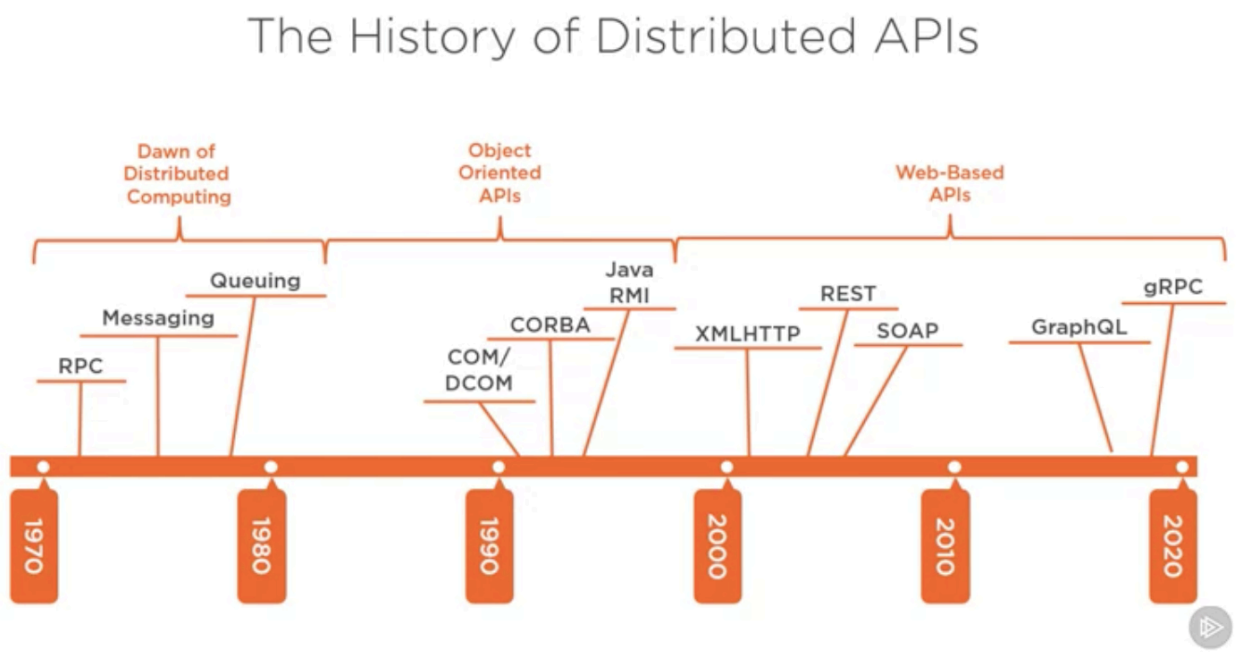
So when you consider creating a web API, I want you to realize that this is a problem that has a long history. We've been trying to connect different systems together for a long time. In the early days, we were just getting started with distributed computing, and so the first ideas of RPC, or remote procedure call, were being invented as early as the early 1970s. We get into an early version of messaging and then even looking at things like queuing. And both messaging and queuing, even though they've evolved over the years, still exist today.

Later in the 1980s and 1990s, we started building object oriented APIs as our programming started to be more and more object oriented. And so you see COM and DCOM come out of Microsoft, CORBA come out of the Java community, and Java and RMI come out of Java central as well. And these were ways to try to create distributed computing around the concepts of object orientation.

During the .com boom of the late 1990s, we started to think about the web as being the way we're going to build APIs, much in the way that you're considering creating a web API to be able to communicate with remote servers yourself. This started with XMLHTTP, which was an early component inside of browsers, and that people started using to do communication between browsers and servers. Soon after REST was introduced, but it really didn't get large adoption until a few years after its introduction. In the meantime, something called SOAP came out of the Microsoft camp as a solution to this. And it tried to solve a lot of solutions and generally has fallen out of favor lately, but you still see it used a lot in enterprises. But it tries to solve some of the same problems that REST is.

More recently, we've seen some new approaches to how to solve some of these problems, like GraphQL and gRPC. For creating distributed APIs that are going to work with communication between apps and servers or websites and servers or even business to business servers, REST still represents a great percentage of the kind of work that's being done, and has reached a level of maturity that's really useful.

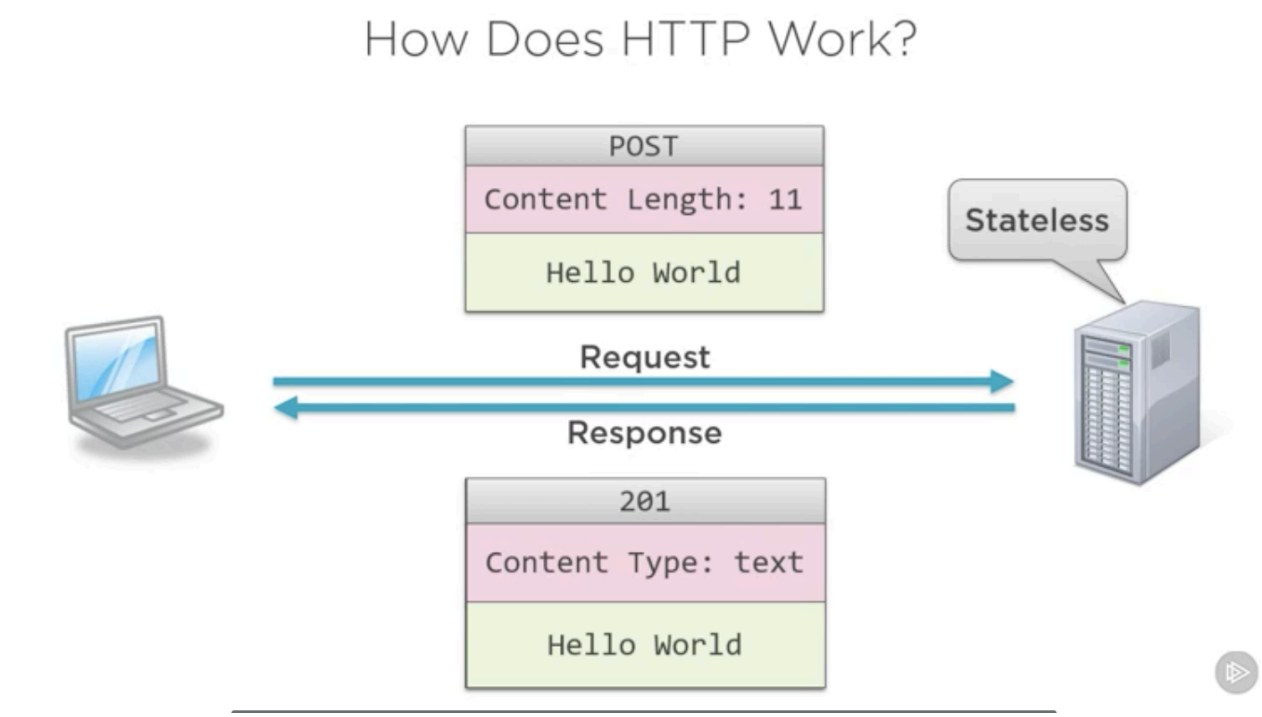
That doesn't mean you shouldn't be looking at GraphQL and gRPC as other ways to solve those problems, but they are segmented for very specific use cases.



So one of the first questions I want to ask before we dig into the actual design is, do you really need one?
Are you building a website?

Are you building a single page app on a website?
Are you creating a mobile app?
And if not, why are you building an API?
I want you to really think about the problem of building a web API requires some finesse and some time to design, as well as implement it. And so make sure you really need to expose an API.

If you really have desktop clients and you're talking within your organization, you might not really need a full featured API, you may need another way of doing that communication. Even things like 3-tier or client server are still really laudable, unless you need the features of using a web API, and these are being able to go across different networks, being able to easily cross things like proxies and firewalls, all of those features represent a good reason to build a web API, but building a web API just because you have a product isn't really a good reason. Make sure you have clients that are going to want to call into your service and that you have some way of justifying the expense and cost of creating, as well as maintaining that API.



Fundamentals of how HTTP works?

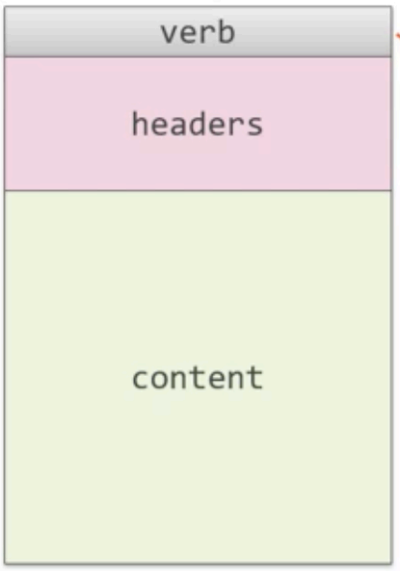
In many ways, we take the web for granted. When we open up a browser, go to a webpage, we expect it to open up and work. What's going on under the covers is a ballet of simplicity. There's a lot of things happening, but the type of calls that are being made are actually really simple and have stood the test of time from their very early creation points.

So let's talk about what happens when we make those sort of requests. So when a client of some sort, say you on your laptop going to Capgemini.com or any web page, it's going to make a request to some remote server.

We're not going to talk about how that address gets converted into a server address, that's not really necessary for this talk, but we have this idea of a request, which is just actually a text document.

This document contains three pieces:-
a verb for what action to take,
headers, which is information about the request,
and then content that is optionally asked for.

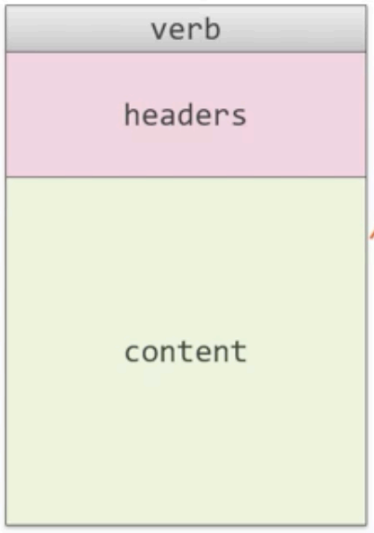
The Request Deconstructed



Action to perform on the server

- **GET**: Request Resource
- **POST**: Create Resource
- **PUT**: Update Resource
- **PATCH**: Update Partial Resource
- **DELETE**: Delete Resource
- More verbs...

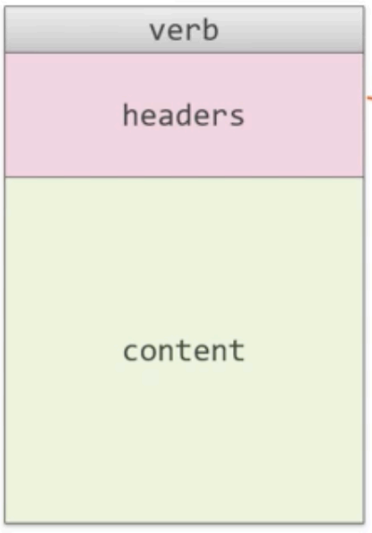
The Request Deconstructed



Content Concerning Request

- HTML, CSS, JavaScript, XML, JSON
- Content is not valid with some verbs
- Information to help fulfill request
- Binary and blobs common (e.g. .jpg)

The Request Deconstructed



Metadata about the request

- **Content Type**: The format of Content
- **Content Length**: Size of Content
- **Authorization**: Who's making the call
- **Accept**: What type(s) can accept
- **Cookies**: Passenger data in the request
- More headers...

So for example, if we wanted to create a new thing on that server, we might post, which is one of the verbs. We have some metadata that says, hey the length of this request is 11 bytes in this case, and then we're sending Hello World as the content we want it to post.

The server gets this request and can either recognize it as something it can do or it might go ahead and reject it, but then it sends back a response. And notice that these are two individual calls essentially. You do not have a connection from client to server that lasts very long. The only way to scale up these sort of situations is that each network request is short lived. You make a request, you get a response, you make another request, you get a response, and so there's a lot of these happening, sometimes even in parallel from your browser. This response is, again, just a typical piece of data. It's just, you can think about it as a textual document that contains a status code, and that status code says, did it succeed? Did it fail? Why did it fail? Any headers that might be useful to you, and then any content that the server wants to send back as part of that response.

One of the important ideas here is that the server itself is typically stateless. So that you're not going to care between these requests that you're stuck onto the same server. A request is coming in, the server is going to try to fulfill it, and then it's going to forget about you as soon as it sends the response.

And in this way, it can handle lots and lots of requests, instead of having all this state that's filling up the memory and disk, waiting for you to make another call. Because unlike having a strong connection to the server, which you might be used to in database programming or game programming, the idea here is that it doesn't know if you're going to make another call because every connection to the server is very short lived.

Let's take that idea of the request document and let's deconstruct it a little -

The **verb** is the action that you want to have performed on the server.

The most common of these is GET. When you want a web page, it goes and says, hey server go get me the home page of Pluralsight.com.

You can also say POST, which is please create that new thing I'm asking you to create on your server, maybe that's storing it in the server, maybe it's starting some process like creating an invoice or making a payment.

PUT asked to update a resource that already exists on that server. There's also sometimes used PATCH, which is to say please update this part of the data of this resource. A common example of PATCH is please update the address of the customer. I don't need you to send me the entire customer just to update the address, so PATCH is going to allow me to do these updates within a resource.

Then there's DELETE, which should be pretty obvious. Get rid of some resource that exists on your server, it may be deleting an object that I didn't mean to create or it may mean that I'm going to unregister with your website, whatever it is, it says delete some resource. And then there's actually a bunch more verbs, but these are the five that are used 99.5 % of the time.

We'll actually see some other verbs like OPTION being used a couple times in our discussions, but these are the five that I would focus on.

#The **headers** are a set of name value pairs that are metadata about the request.

So some of the common ones are what type of content is the content section holding? It is binary data? It is a JSON file? Is it an XML file? Is it plain text? This is to tell the server how to deal with this request.

The content length is there to hint at the server how much content there is. So it doesn't go ahead and not know how the end of the content is as it's pulling data across the wire.

Authorization headers are there to says who's making the call.

Accept headers are there to say when you send me a response, what kind of data can I accept?

Cookies, this is data that's being sent with the request that expects the server to also pass back as a way to have state through the entire process.

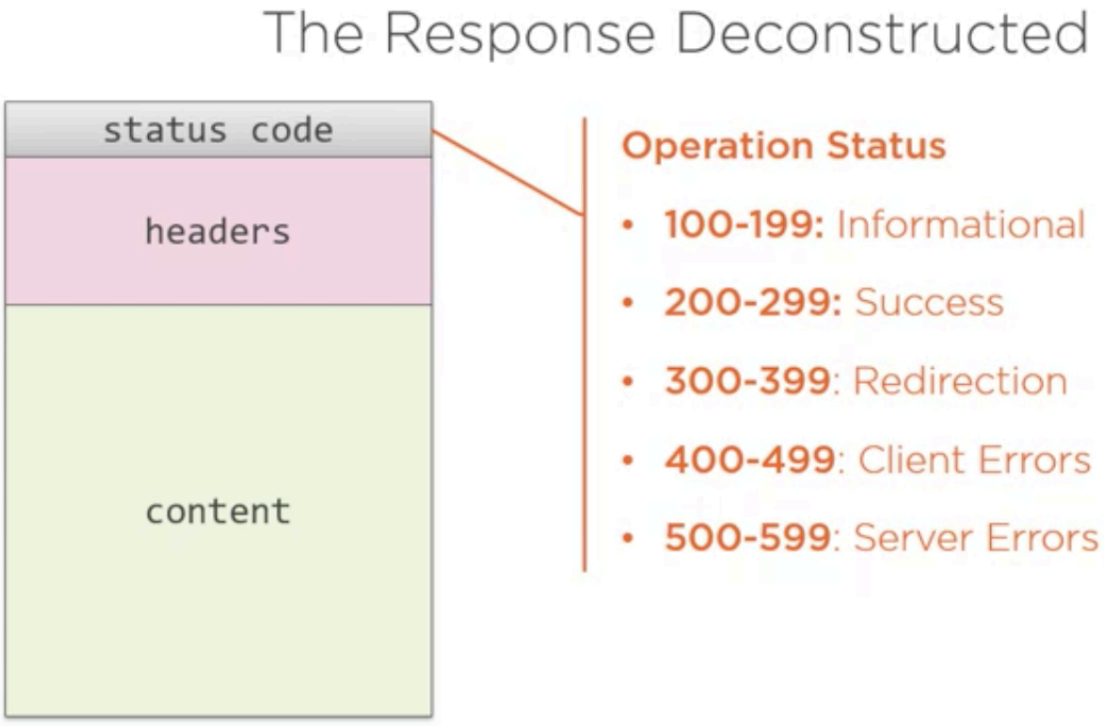
And more, there's literally hundreds of headers, and you can define your own headers, if those are useful to you, but for the most part, these are the ones you're going to want to think about.

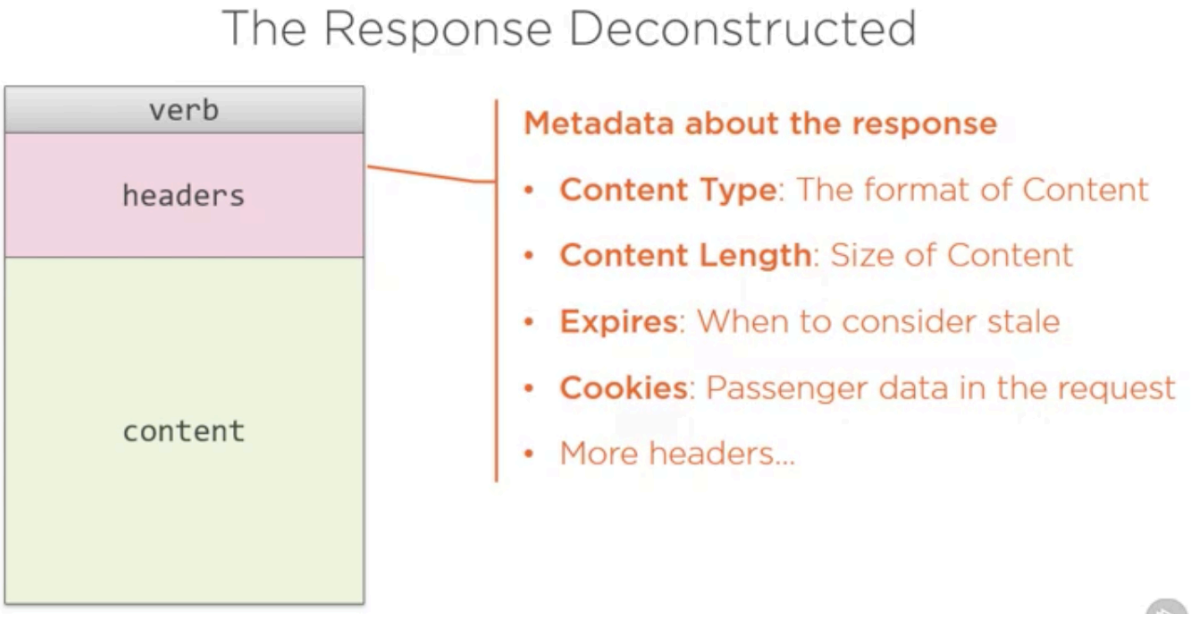
Finally, **content,** this could be anything.

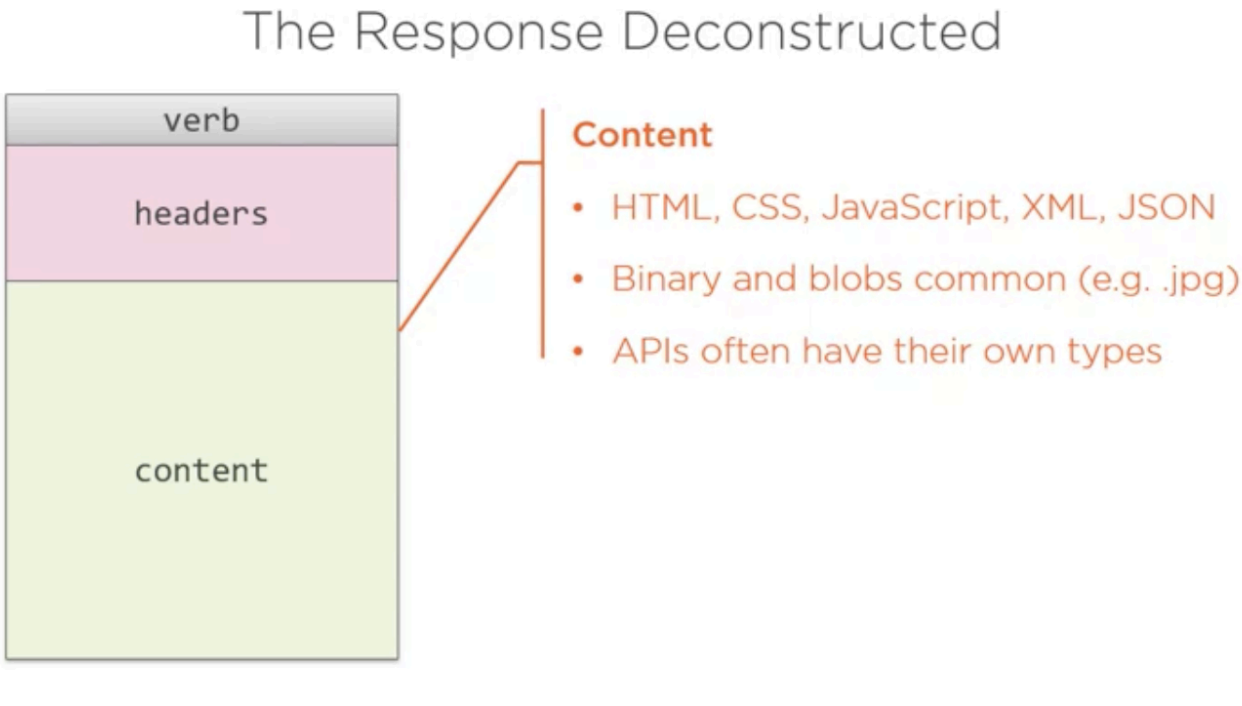
So if we look at a simple web page that you're going to request, you're normally going to request and it's going to return an HTML page, and then that HTML page might have CSS and JavaScript or JSON requests in it that are then also requested in the same way, and that's why each of these requests are discrete. So a simple HTML page may have dozens or hundreds of requests. The content for a request isn't valid on some verbs. The most of these is GET.

GET never has a content body because GET is just requesting that you return something that it wants. So as per the spec, you can't really send any information about what you want, as content has to be in the URI or in the headers.

Let's do the same thing with the response:-







The **status code** is simply a number that represents what kind of success, and it does this in a series of ranges.

100 to 199 are informational, you'll rarely see 100 status codes, but they do exist.

The most common you're going to see is 200 to 299, and these are status codes that say, hey what you asked for, here it is, and I was able to do it successfully, 200 being the most common of these.

300 to 399 are returned when what you requested for needs to be gotten some place else. So this is how you can say hey you need to go to this new place for your web page or this is data is already cached for one reason or another. **So these are about redirection to different sites.** So

when you think about redirection, it isn't saying it succeeded in getting you the data, but it's not saying you failed in requesting it, just that you asked for it in the wrong place.

400 to 499 are errors. These are typically errors in the way you made the request. You may have not included a query string or some part of the URI that makes sense, or simply the URI doesn't exist. The famous 404, of course, is in this range.

500 to 599 are errors that the server has, i.e. something went wrong on the server, it's not your fault, it's our fault.

Same idea here with **headers**, they're very much similar to what we're going to see in the requests, except we're going to tell it this is the kind of data it is that we're sending back to you, this is the length of that data, this is how long that data should be cached. So the idea of expires is that the browser or some other thing might cache it for 10 minutes, 30 seconds, 2 years, because it's not apt to change. And then cookies are being sent back normally from the cookies that are being included in the request. Though there may be more cookies included on the server that are then expected to be passed in subsequent calls.

And finally, the **content** is very much like it is on the request, it might be HTML, CSS, might have images like blobs, or APIs can often have their own types as well.

What does the term REST mean?

It is actually a pseudo acronym for REpresentational State Transfer.

The idea here is to have transfer of data or state be representational of the kinds of messages that you want to use. And so these concepts include separation of client and server that all the server requests should continue to be stateless, that all the request you make should support cacheability and that they're all going to use a uniform interface or a URI. And all this comes directly from the source of REST, Roy Fielding's doctoral dissertation. So from that dissertation, the fundamentals of what it really means to be RESTful are brought forward.

But we're going to talk about, again, the pieces of REST that are going to benefit you to create great software that's going to live for a long time. The reason we're not going to go deep into making it perfectly or dogmatically REST is that **REST itself has some problems.**

Sometimes it's too difficult to make sure your application is completely RESTful. There's some concepts that can get in the way of making sure that your API is completely qualified as REST. This sort of dips a toe into the idea of the dogma of REST versus the pragmatism. I'm very much on the pragmatic side.

The differences here is that REST can provide a structured architectural style within the needs to be productive in your work life. This means that as we design our API, we may be shortcutting REST here and there, but only when it benefits the software that we're building.

WELL-DESIGNED API

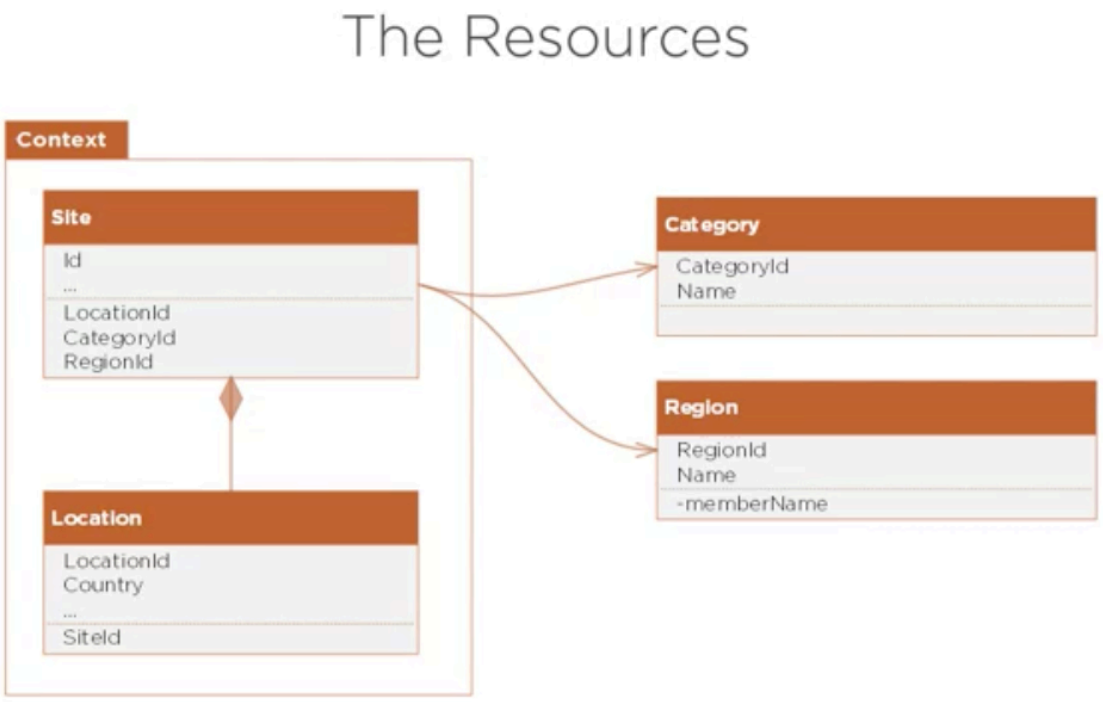
Design Your API First

- Can't fix an API after publishing
- Too easy to add ad-hoc endpoints
- Helps understand the requirements
- Well designed API can mature

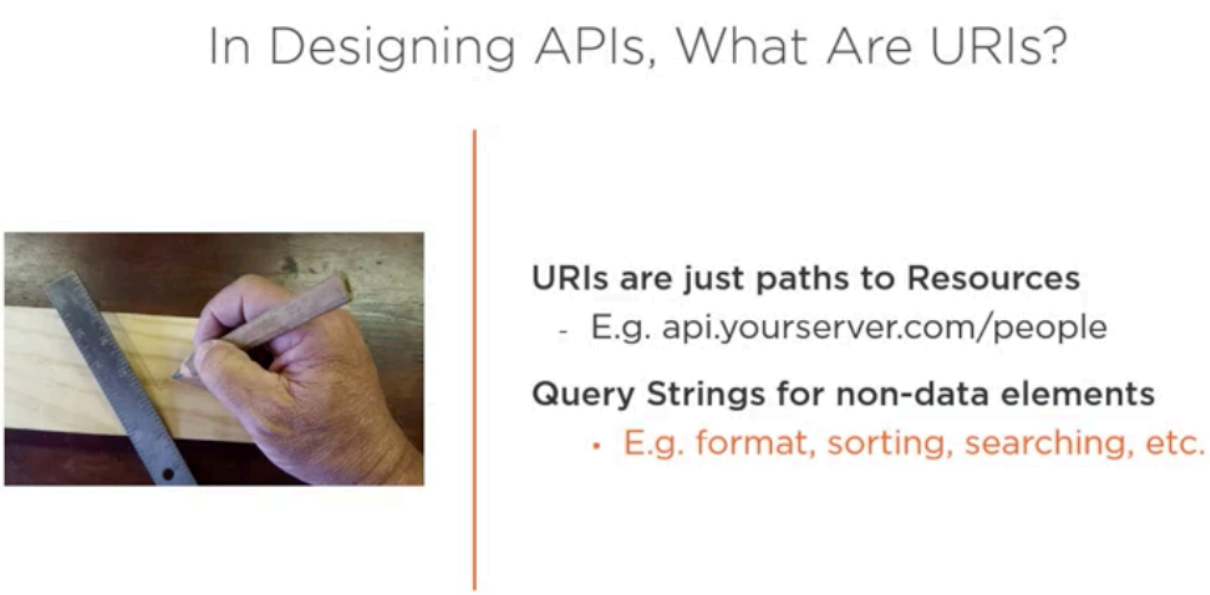
For REQUEST

For RESPONSE

Resources



URIs



API Design

- Nouns are Good, Verbs are Bad

```
/getCustomers
/getCustomersByName
/getCustomersByPhone
/getNewCustomers
/verifyCredit
/saveCustomer
/updateCustomer
/deleteCustomer
...
```

API Design

- Prefer Nouns

```
/customers
/invoices
/products
/employees
...
```

In well designed API:- " Nouns are good, verbs are bad ! "

Identifiers in URIs

- Use unique identifiers
- Does not have to be 'primary keys'

```
/sites
/sites/1
/sites/stone-henge
/sites/uk-101
...
```

IDENTIFIERS

Query Strings

- Use for non-resource properties

```
/sites?sort=name
/sites?page=1
/sites?format=json
...
```

QUERY STRING

GET

- Retrieve a resource

POST

- Add a new resource

PUT

- Update an existing resource

PATCH

- Update a resource with changes

DELETE

- Remove the existing resource

VERBS

How do I do Verbs then?

Resource	GET (read)	POST (create)	PUT (update)	DELETE (delete)
/customers	Get List	Create Item	Update Batch	Error
/customers/123	Get Item	Error	Update Item	Delete Item

Idempotent

adj: operation that can be applied multiple times without changing the result

Idempotency in REST

- Operations result in same side effect
  - GET, PUT, PATCH and DELETE
- POST is not idempotent

Let's assume case of a PUT operation, you made a change and it says in postman that changes applied successfully as you see the updated output now.

Now if I run the same PUT operation, it shouldn't show me a result saying that changes weren't applied successfully. Even though we didn't change something, it should show the same list to us. It is what a well designed api does!

DESIGNING RESULTS

Decide Formats During Design

```
// Abide by Accept Header:
Accept: application/json, text/xml

// Return sane default (usually JSON)
Content Type: application/json

// Prefer not to use query strings for formats
/api/customer?format=json // <- Antipattern
```

COMMON FORMATS

(CAN BE SELECTED HOW TO FORMAT RESULTS IN HEADER IN POSTMAN, ALSO CAN TYPE VALUES IN KEY-VALUE PAIRS)

Common Formats:

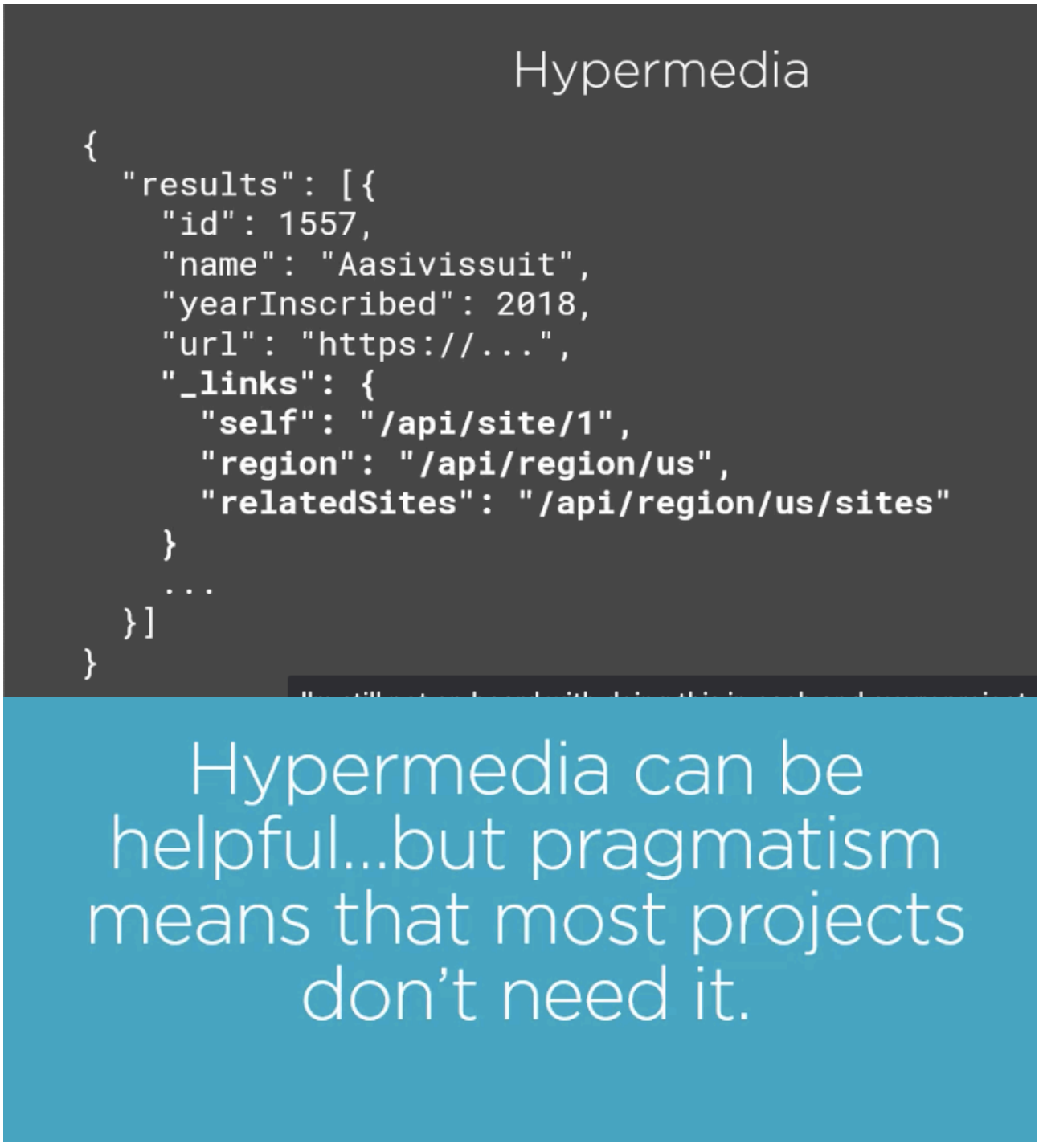
- JSON: application/json
- XML: text/xml
- JSONP: application/javascript*
- RSS: application/xml+rss
- ATOM: application/xml+atom

* Requires callback parameter

HYPERMEDIA

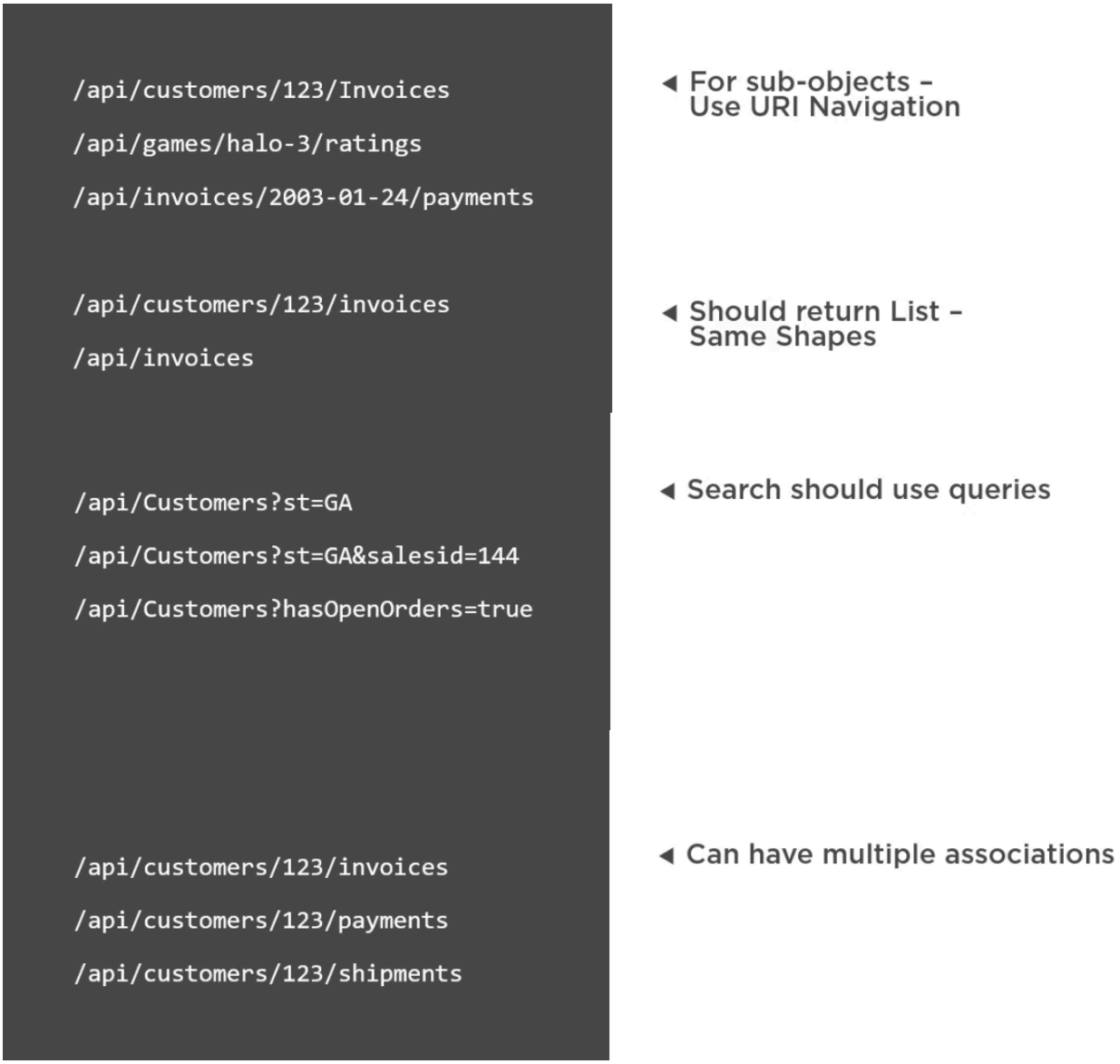
Hypermedia

- Allows results to be self-describing
- Allows programmatic navigation
- Adds complexity



ASSOCIATIONS

When I talk about associations, I typically mean sub-objects for existing APIs. For example, I might have the invoices related to a specific customer, I might have the ratings related to a specific game, I might have the payments related to a specific invoice.



The idea behind associations in these simple cases is to allow the navigation of the URL to imply ownership. You would never use this customers/123/Invoices URI as a way to look at all invoices, it's clear that this should be the invoices of that specific customer. In the same way, both of these URIs, one that returns all or individual invoices, should also return the same type of shape that the invoices within a specific customer would return.

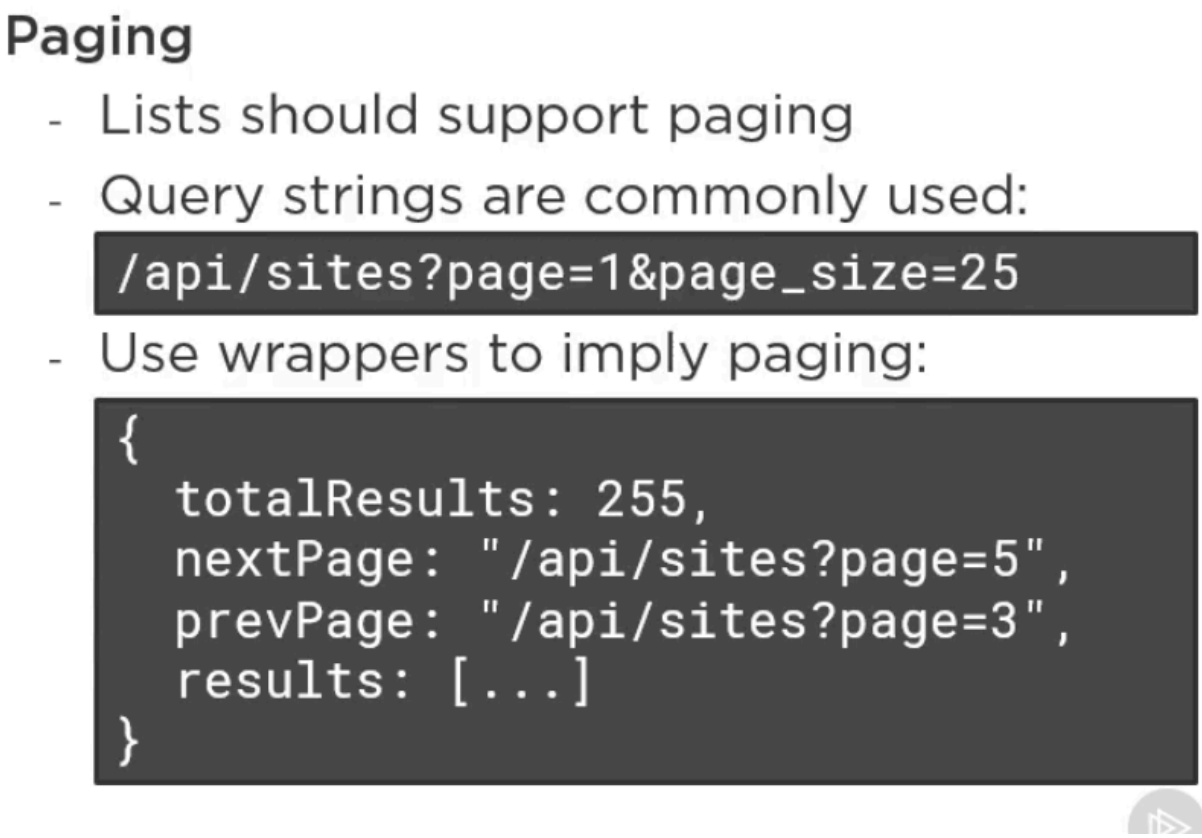
The difference here is the scope. The invoices inside an individual customer should return a collection, but only invoices that belong to that customer. Where the other URI, a top-level URI, would return invoices across different customers. In this same way, URI endpoints can have multiple associations. So even though we have invoices for a specific customer, we could also have payments for a specific customer or shipments for a specific customer. There's not a limitation that an endpoint for, let's say, a particular customer or a particular resource, has to have only one association to it. It's pretty common for it to have multiple associations. These associations shouldn't be confused with search queries.

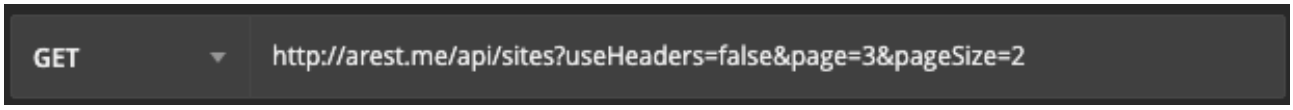
PAGING

Most lists that you're going to be dealing with inside of your API should probably support paging, and the idea here is pretty simple, that unless your list is a very concrete list, let's say you have an API to show all of the countries in the world or all the states in the United States, etc., those are pretty fixed lists that you're going to always want to return as a block.

But anything else in your system, especially typical day to day resources, you're going to want to support paging for a few reasons. **One, to make sure clients don't pull back everything just because they don't know any better, but also b) to allow you to have the interactions with users that make more sense to them. Query strings are commonly used to do this paging.**

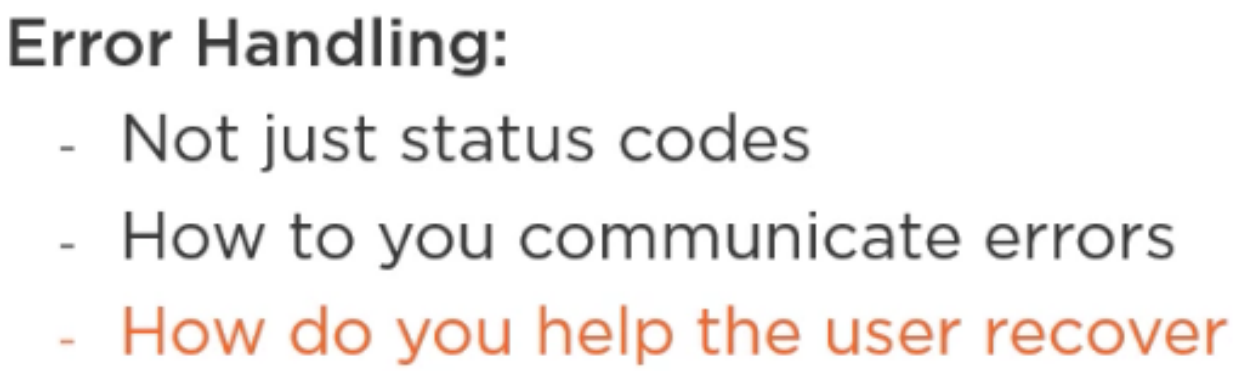
Some people do use URI sections to do this, but I almost always suggest query strings because query strings are saying we want to do something different with an existing API. We'll often want to wrap our results inside of a wrapper section in order to give this information. **And so having an API that specifies what page they're looking for, as well as optionally a page size, is a pretty typical way to go.** You may want to control this and always have the same page size, but I find using the page size ends up being pretty simple. And then you can simply return, as part of the results, the next and previous page, as well as the total results, as that's something that's very commonly shown to the user, so they don't just go next, next, next, next, and realize they have a





million records that they might be paging through. They might change their query instead. If you've ever done a Google search with a billion results, you know that adding terms to the search is often a better solution.

ERROR HANDLING



In your API, you certainly can just return status codes, and most developers will be able to make sense of that, but it's often helpful to allow you to communicate information about the errors.

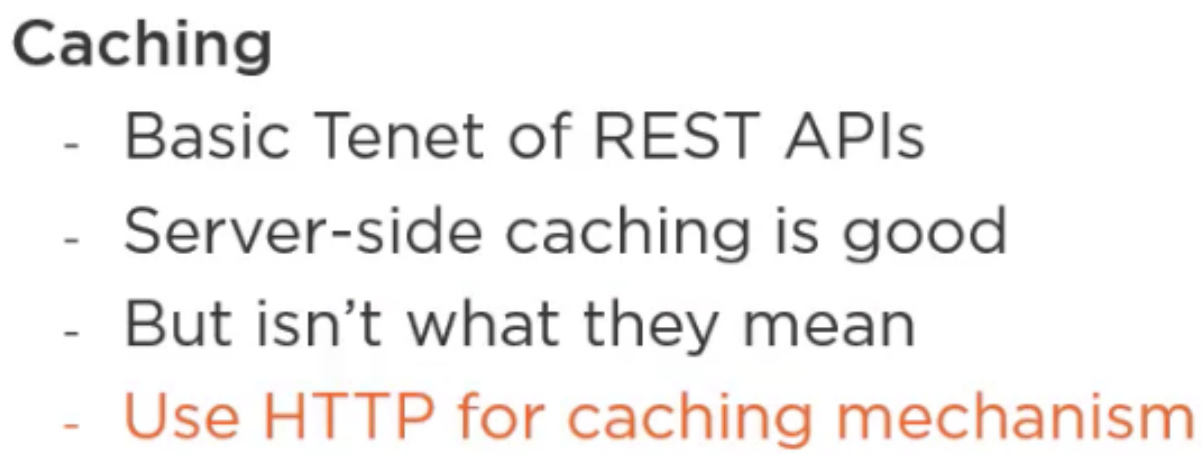
You want to use error handling to help your users recover from errors that are related to them. So if they've made a mistake, we want to be able to communicate that something's wrong with their request. So often this is returning the object with the error information. So you might return a bad request because they've made a mistake, but also something about the error to say how to fix it. You failed to supply the id, therefore you want to fix that and try to resubmit it.

◂ Return object with error info

◂ Not necessary for obvious errors

Now this isn't always the best idea. In the case of security concerns, you'll often want to just return the failure without an indication. What I mean by that is if you have an API that does login and they've supplied a username and password, you might not want to return bad password, didn't match with username, or username doesn't exist. Those can give users of those APIs a false sense of what is wrong, which is a security problem for you, because that means once they start hitting correct user ids, then it becomes a little simpler for them to just barrage you with passwords, hoping to get the right one. And so use this wherever you think it's helpful to the user but doesn't compromise your own security. There are cases where you don't need to return a body because the error is just obvious on its own. The most common of these is file not found.

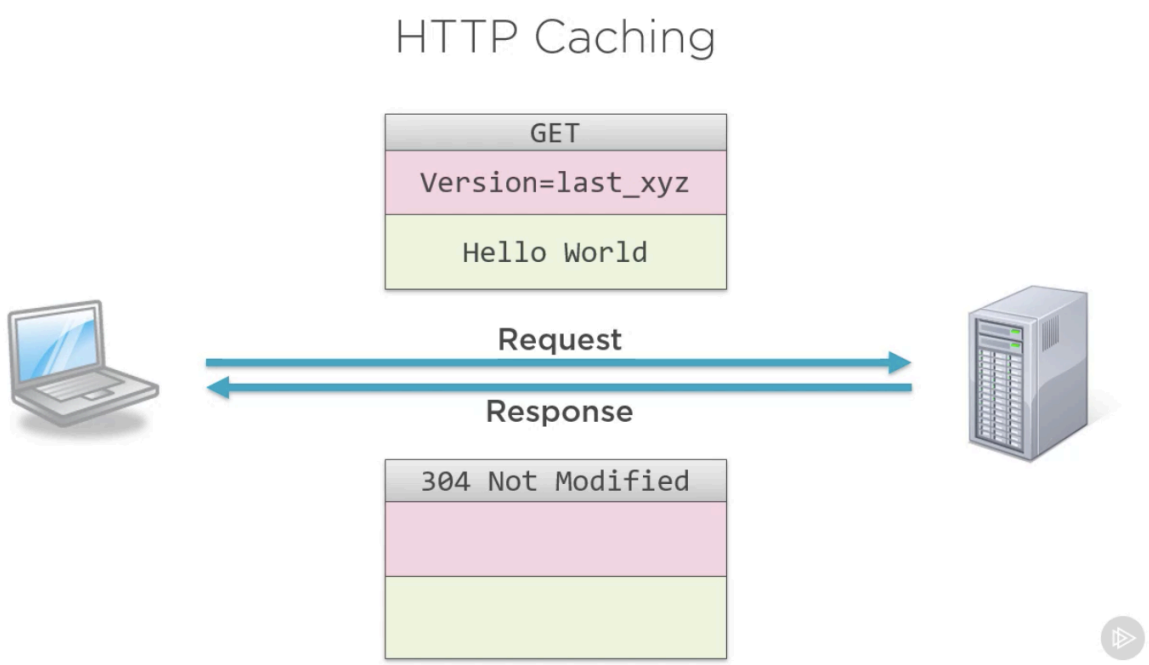
CACHING



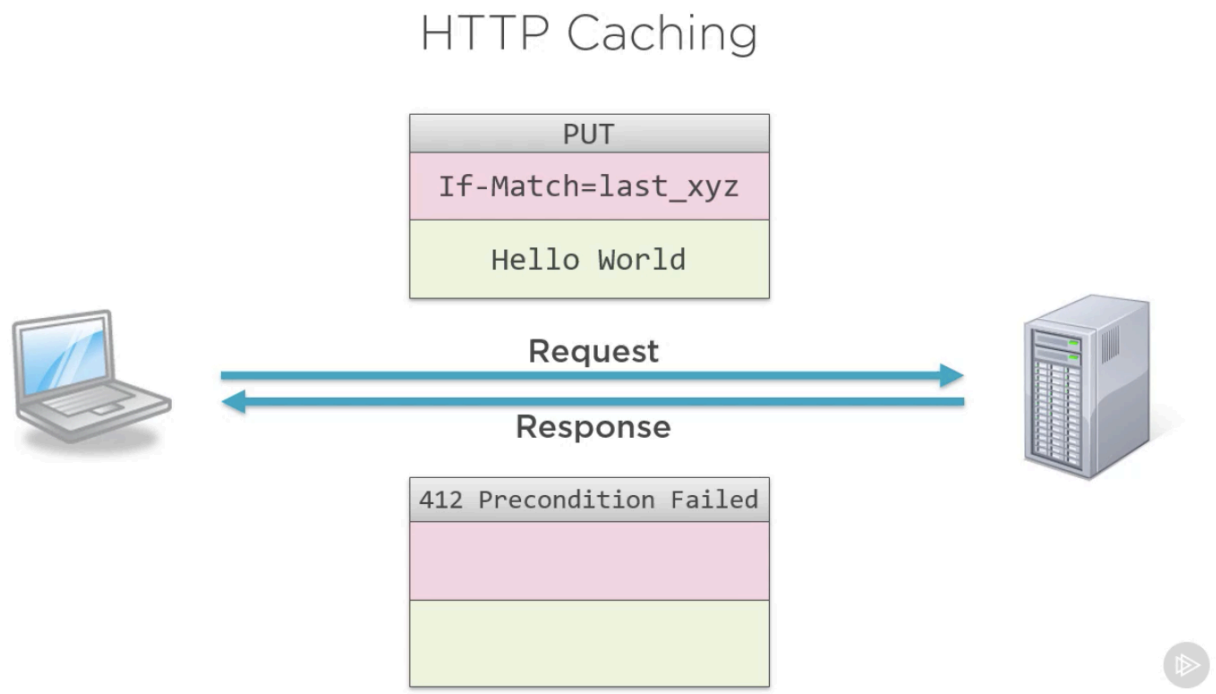
Caching is one of those ideas that is required to be truly RESTful. It's a basic tenet of the way that you would build REST APIs.

Now not every API needs to have caching, but in order to scale up and be really useful, you should think about designing caching into your API. While supplying server-side caching of results is good, that's not really what REST is talking about. If four people ask for the same customer, you've cached it on the server, so you can return it faster, awesome. That's not really what the REST API needs to worry about for caching. What they mean in these cases is to use HTTP for caching. Now what does this involve?

When you make a request over HTTP and they ask for something and include the last version that they were given, the response should be a Not Modified. And so in this way, the client can ask the server, do I have the latest version, without the server having to find it and then send it back so you can compare it. So in some ways this really is versioning of individual instances of data or resources that you're returning from servers.



Another way it does this is on the request it can use a header called If-Match. So I retrieved this object and got a magic number or magic identifier to say this is the version I have on the server, and then when I try to go update it in a way of doing concurrency, I can say, hey, make these changes, put these fields in the right place, if your version is the same as my version. And if there is no match, if it has been updated on the server since I retrieved it, the return code that you can expect to send back is precondition failed, because the header here becomes a precondition.



A great way to handle this caching is something called entity tags, or ETags for short. They support strong and weak caching, so basically the idea would be is this a version tag for this payload that you can hold onto for a long time or a short amount of time?

Strong Caching means I can go ahead and store it in my rich client or on my phone app and 4 weeks later I should be able to send it back with changes and this ETag should be able to be constructed by the server to see whether there's a change.

Weak caching support is for things that are just very short lived. These ETags are returned in the response and we can see ETag as a header type with some identifier, in this case it is a unique string that represents the version of the resource on the server. And if you want to return a weak type, you start it with W/ and then use that same format in an ETag. You want to indicate to the developer how strongly this caching support is included.

Essentially this means that if you're doing a GET, you should use If-Match, and to indicate 304, if it is already cached, and don't return it back. So like we saw in the other example, if I request it and say do a GET, but only if it's not the version I already have, and that's what the If-None-Match means, all that gets returned from the server is not modified. So the version you have is the latest version I have. And so that server communication can become very cheap because it should be a very quick roundtrip. Similarly, on PUT and DELETE, you're going to indicate 412 if the version they have is not the same. So this is different. On a GET, you're going to return a 3 or 4 if it is the same and for PUT and DELETE, you're going to return a 412 if it's not the same. And so, for example, here is a PUT, we can say please update these fields if it matches this and if it doesn't match it, so we didn't actually have success in executing the PUT or DELETE, we can say precondition failed in this case. Now because it's in 412, it's in the failure part of the status codes, so whether you look for 412 specifically or you just show failure, you're still doing the right thing.

Entity Tags (ETags)

- Strong and Weak Caching Support
- Returned in the Response

```
HTTP/1.1 200 OK
Content-Type: text/xml;
Date: Thu, 23 May 2013 21:52:14 GMT
ETag: "4893023942098"
Content-Length: 639
```

- Request with If-None-Match

```
GET /api/games/2 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8863
If-None-Match: "4893023942098"
```

- Use 304 to indicate that it's cached

```
HTTP/1.1 304 Not Modified
```

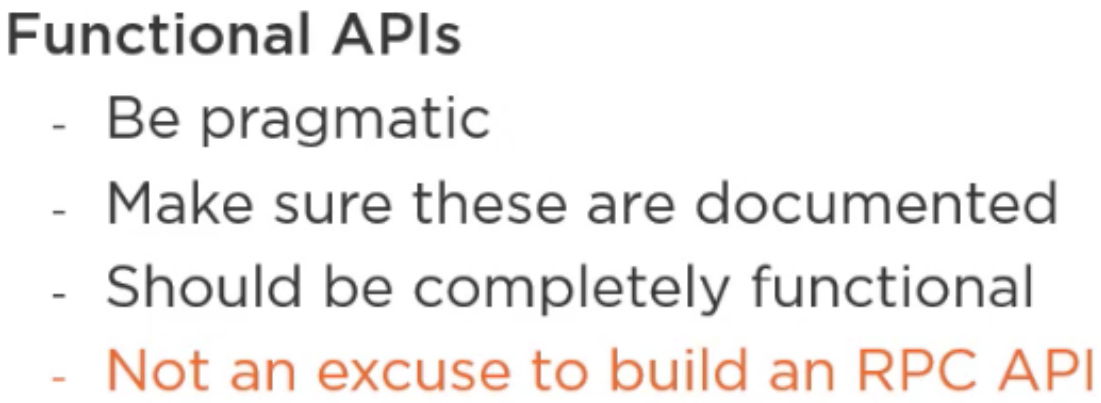
- For PUT/DELETE

```
PUT /api/games/2 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8863
If-Match: "4893023942098"
...
```

- Use 412 to indicate that not same

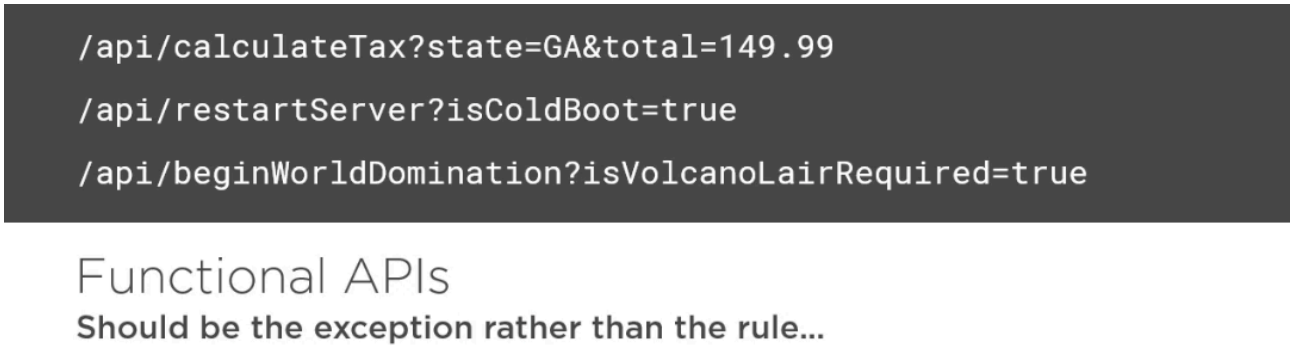
```
HTTP/1.1 412 Precondition Failed
```

FUNCTIONAL APIs

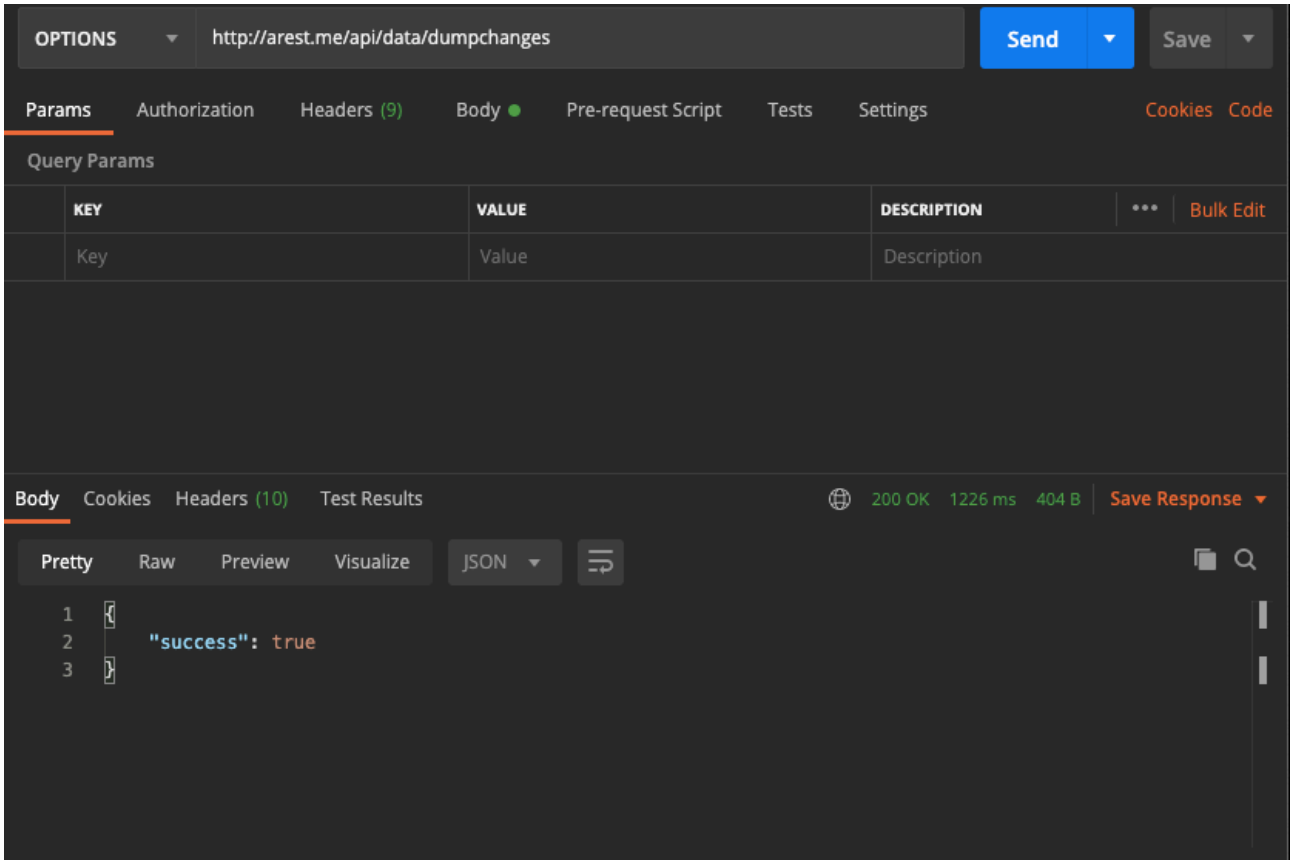


So ultimately when I'm building a RESTful API, again, I want to be pragmatic about it. Ultimately the goal of a REST-based API is to fulfill some set of requirements, some needs, some customer requests. And so functional APIs are not terribly RESTful. But occasionally, you know, once or twice in any large project, you're going to need some call to the server that needs to do something that is functional. **Restart a machine, recalculate totals, whatever the case may be.**

There are these one-off operational APIs that you're going to want to support, but aren't truly necessarily RESTful. Now what I find some people doing to make sure that they stay perfectly RESTful is they'll build a whole new sort of system to handle these operational things, but most projects only need a small handful of them. So why not bake them into your RESTful API? Because of their very nature, you're going to want to make sure these are well documented so people can understand the side effects because almost always these APIs will have some side effects. Make sure they're completely functional and that you're not starting to build an RPC layer over REST just because you don't understand what you're trying to accomplish with REST. So typically functional APIs should be the exception rather than the rule. If you find yourself designing dozens of these or even as little as 10 of these, you're probably trying to get around the limitations of REST and you need to go back and look at your design.

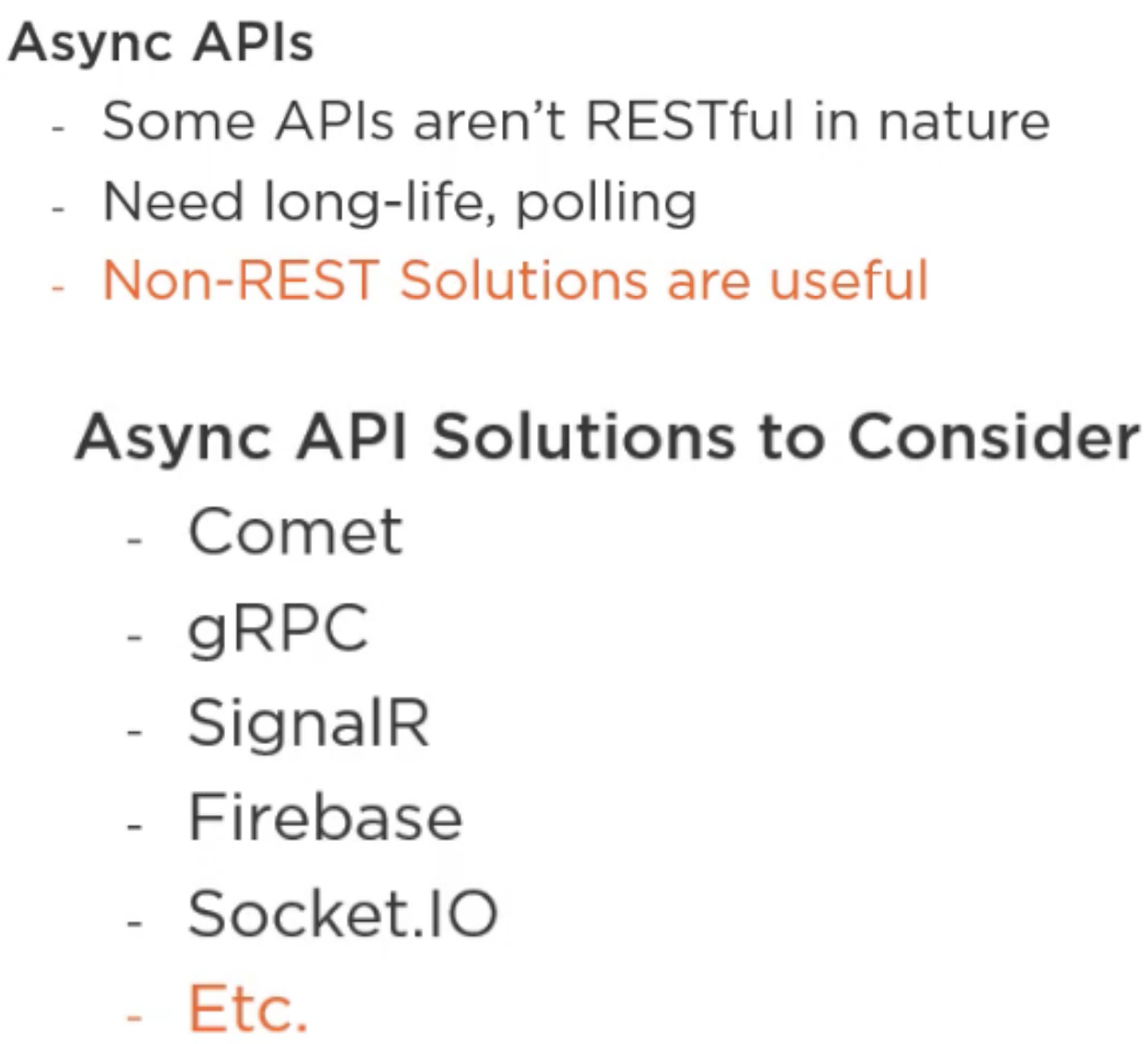


So here's an example of calculateTax, I'm going to send you the state and I'm going to send you the total and you would return me back because you have implicit knowledge on the server about what those state requirements about a tax would be. Or in this case, restart the server. Start a cold boot because you're going to allow some server to be restarted because you have found some functional issue. Of course, one of these might be very useful to your users and one of these might be very useful to your IT team. And, of course, the last one here, I would not put it in every system, but when you need it, it's really crucial. When you want to begin world domination, determine as an optional whether you need a volcano layer or not.



\\ example of a functional api to dump any changes done while working on api \\

ASYNCHRONOUS APIs



So there are times when asynchrony in an API is something you're going to want. The fact is that some APIs aren't RESTful in their very nature. You can imagine things that are polling data, that are communicating like a two-way chat system, or a notification of changes on the server. They often need long-life or polling or other systems to have these asynchronous long-lived connections to a server that just simply aren't RESTful.

So we need to start looking at non-REST solutions when you need these long-lived APIs. This does mean implementing something that is separate from your RESTful API. I mention this because I find a lot of people try to shoehorn these into REST and I don't suggest it.

Some async API solutions to consider are Comet; gRPC, which is something that's fairly new out of Google; SignalR, which is a .NET specific solution, but one that scales really well with Azure SignalR; Firebase; Socket.IO ; Etc. Even just using Web Sockets to do that communication can be helpful. So that when you need these long-lived connections to a server, don't try to implement them inside of your RESTful APIs, know that you're going to need to find a separate solution, even though those solutions may be very much side by side with your RESTful API.

SHOULD YOU VERSION YOUR APIs?

It's a good thing to think about because once you publish an API, it is set in stone. Once clients have written against the API, you're going to have to maintain that version of your API. And once you have users that rely on the API, you're not going to want to change it willy-nilly. You're going to want to make decisions about how to handle changes in your API without breaking the clients.

Now if you have an API that's only being used by your own team, maybe versioning is overkill. But as soon that API has internal or external customers, if it has users that are actually writing code that may not be just your small team, versioning in an API is super important, and it's easier to handle during the first version of your API than trying to shoehorn it in later.

Ultimately your API will have changes in requirements no matter what you do, no matter how well you've designed it, and so your API is apt to change. The goal here is to evolve your API without breaking existing clients. Now we don't mean that every time you release a product version that you're going to change the version of your API. They can be decoupled. Tying them together is just confusing, especially if the API doesn't have actual changes to it. (API version isn't Product Version)

API versioning is harder than product versioning. The reason for this is you need to support for old and new versions and to have a story about deprecation over time. In theory, you can have side-by-side deployment of multiple versions, but it usually isn't feasible. You usually want some indication from the users about what versions they want to use. And allow older code to not have to make changes to continue to use an API unless you truly do want to sunset those APIs. Ultimately the decision is up to you, but planning for and designing versioning up front will make maintenance of your API easier longer term

VERSIONING AN API

There are a lot of ways to version an API. Not all of them are ones I would recommend. You're going to want to find the mechanism or the way of doing versioning that makes the most sense to your organization. And this is going to depend a bit on the requirements. Ultimately you're serving your clients, not yourselves. Making versioning easy to use for your clients is way more important than making it easier on your development staff.

The first option for versioning that I will talk about is one that I see pretty commonly used, and that is versioning in the URI path.

```
// URI Path
https://foo.org/api/v2/Customers
```

Versioning in the URI Path

Pros:

- Very clear to clients where the version is handled

Cons:

- Every version needs to change URIs, can be brittle

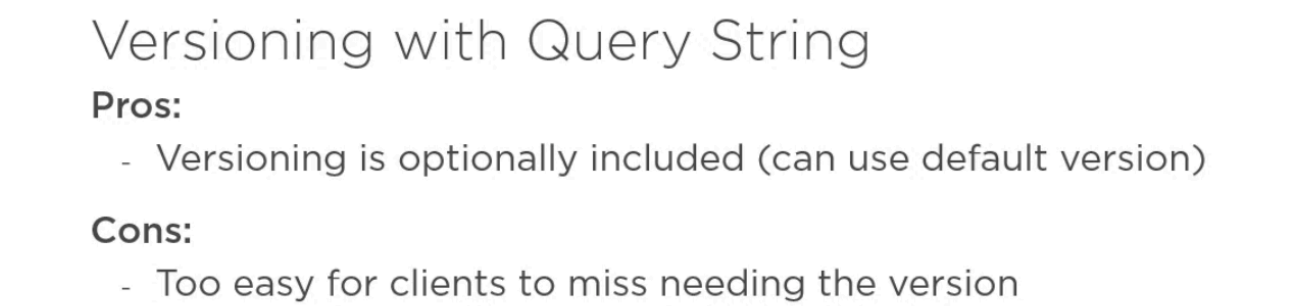
We can see in this example that the v2 inside the path to our customers API is indicating what version of the API. While it's very clear to clients which version they're using and how it can be handled, it tends to be brittle. Every time you have a version change, you're going to need to have clients change all the URIs. And so usually I don't recommend this path.

Another versioning strategy is query string versioning. In this case, we can simply add a query string that asks for a specific version of our API.

The benefit here is that we can have a default version that will always be used by our API when the version query string isn't supplied and then simply add it to other versions. This often means that newer versions of the API may need to include this every time, depending on where you want to do your versioning.

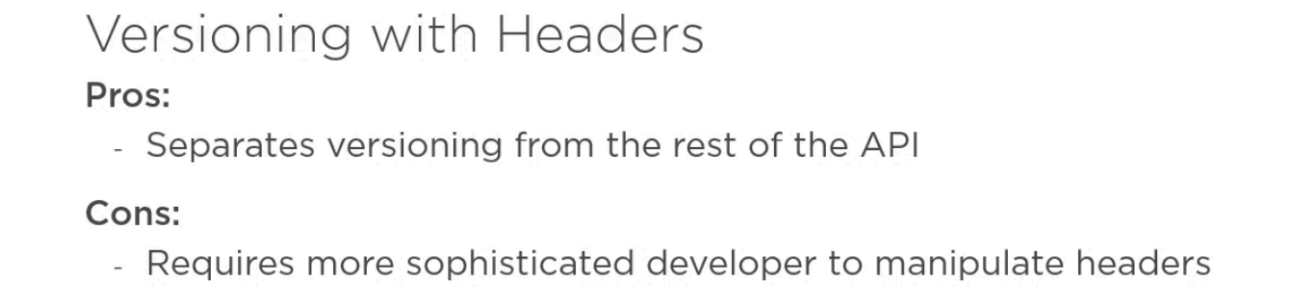
The big con here is it's too easy for clients to miss needing a version. So helping those developers that want to use your API to remember what version to go for tends to be problematic.

```
// Query String
https://foo.org/api/Customers?v=2.0
```



The next type of versioning is something called versioning with headers. You can see the X-Version here is a header that specifies what version of an API to go after.

```
GET /api/camps HTTP/1.1
Host: localhost:44388
Content-Type: application/json
X-Version: 2.0
```

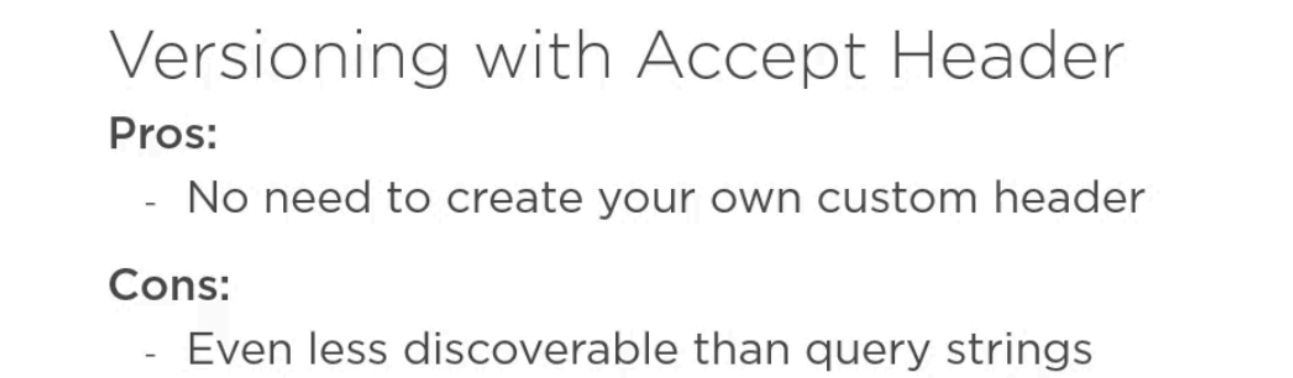


And what's interesting about this approach is that it has some of the benefits of query strings, but it does separate versioning from the rest of your API so that the API writers don't necessarily have to change it, only an interceptor in writing that API needs to put this version in the header. So it becomes a little decoupled with the actual API calls.

The issue with using headers is it often requires a more sophisticated developer, one that knows how to add headers or how to intercept those calls so that those headers can be added to their client code.

You can also version with an accept header. This is beneficial because you're not creating your own custom header, but instead you're using the accept header itself to ask for a specific version of your API.

```
GET /api/camps HTTP/1.1
Host: localhost:44388
Content-Type: application/json
Accept: application/json;version=2.0
```



In this example, we've seen that even though we're accepting application/json, we're specifying the version in that accept header. And the benefit here is the accept header also will keep the version that the client uses so that when the content type is sent back to the server, it can include that version.

You're having the separation of versioning with both the content they're dealing with and the API calls. But it is less discoverable than query strings and also has the same problems that simple headers do, and that is, requires a little bit more sophistication to use.

The last one I'll talk about is versioning with content type. And this is the most complex of them all to implement. But if you need it, it's really useful.

The idea here is that you would include a custom content type. The spec calls in content type to allow you to do vnd. and then your application name as a special type of content. In this case, you could use the accept header as well as the content type with a version embedded in it in order to specify the kind of data you're looking for. This gives you much more granularity to dealing with versions because then part of your application could be in one version, another part could be in another version, and this is especially useful for long-lived applications. What I mean by long-lived is if I go ahead and get a list of customers on a Tuesday and then 2 weeks later I want to update one of those customers, this will tell me that the version of the API hasn't changed because the content and accept header, which usually are married with the content that

I'm storing, should tell me what version of the content that I received from an API, not just the version of the API I'm using. It does require a lot more development maturity to create and maintain the sort of code, but it is very powerful.

```
GET /api/camps HTTP/1.1
Host: localhost:44388
Content-Type: application/vnd.yourapp.camp.v1+json
Accept: application/vnd.yourapp.camp.v1+json
```

Versioning with Content Type

Pros:

- Can version the payload as well as the API call itself

Cons:

- Requires a lot more development maturity to create and maintain

APIs and Security

Do we need to secure our APIs?

An important decision you're going to have to make during the design of your API is whether you need to secure your API and how you go about it. In a lot of ways I've talked to developers who thought their website is small or doesn't deal with money so I'm not going to worry about securing my API. And I want you to think about some key things before you decide not to secure an API.

Do you need to secure your API?

Are you...	Secure?
...using private or personalized data?	Yes.
...sending sensitive data across the 'wire'?	Yes.
...using credentials of any kind?	Yes.
...trying to protect against overuse of your servers?	Yes.

Threats to APIs

When we think about threats to our API, we're often focused on the mysterious hacker, but often threats come from different places. There's eavesdroppers that are listening to or what are often

called man-in-the-middle attacks. So you're going to want to worry about the security, not just of log in and log out of your system, but transport security. There's also securing access to the actual boxes you're dealing with, and while cloud can minimize some of that, protecting your API from physical intrusion and protecting it from employees of the company you work for from getting into it is another thing you're going to need to be worried about.

Users and hackers are often the sort of last line of defense that you're going to want to worry about, but make sure that you're taking the entire threat model here into concern.

So when I talk about securing or locking down your API, what I'm really talking about is security at your server infrastructure.

Companies have spent millions of dollars to make sure that they have these deep authentication systems and then there isn't a lock on the server room. Don't be that company. You're going to want to **secure your APIs in-transit.** We're going over the internet typically, and so using SSL in every single case is important. You can't assume that the quality of your data isn't worth the cost of SSL, not just the $20 certificate or even the free certificates, which are really common these days, but also the computational cost of encrypting and handshaking is almost always worth the expense.

Then we get down to what you probably think of as **API security, cross origin security,** so that not any website can go ahead and start writing code to your website.

Authorization and authentication, this is where you're going to use credentials to not only identify who you are, but what you're allowed to do. And so when I start speaking to developers, they jump all the way down to the bottom of this list, without taking any concern for anything else on this list. And so I really want you to think differently about this when you're going to design your system. ***You're taking the time to design and implement a complicated system to meet customer requirements. The last thing you want to do is leave holes out there so that you spend your nights and weekends trying to patch security holes instead of spending time with your family.***

CROSS DOMAIN SECURITY

The idea behind cross domain security is to prevent cross-site scripting from occurring. In essence, this is a limitation of the browser (**so that not any website can go ahead and start writing code to your website.)**

Most platforms, when you're building an API, don't allow a separate domain to call an API by default. You have to ordinarily go through some hoop to allow another website to call your API. But this is only limited by the browser.

These are browser limitations that are there to prevent malicious code from running in the browser that might call the separate API. This is important to consider because it's different whether it's a public or a private API.

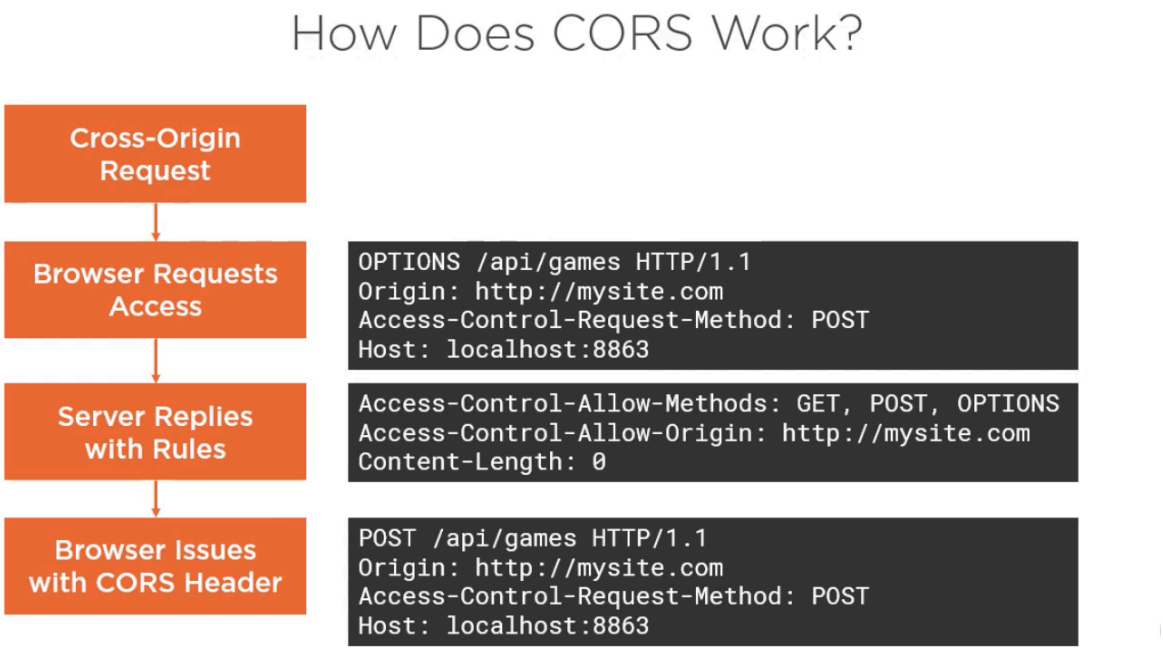
You may never want to allow this if the only people calling it are your individual web pages. If you're building an API for your enterprise app that is only going to be used in the browser from within the same domain, the browser is going to do the right thing and allow it. But if you're building a public API, you should allow cross domain, and for private APIs you should consider it, usually for partners. The standard for doing this is something called Cross Origin Resource Sharing, or CORS. This allows you to have fine grain control over who has access from a browser to your application. So allows you to limit what domains can get in, what resources on those domains they have access to, and which verbs can be executed on them.

For example, you might allow your partner domain to call your API to get to the list of products in your product database, and allow them through GET to only read that product database.
It's important to know that none of this matters if it's not from a browser. If you're creating a mobile app, if you're creating a desktop app, cross origin resource sharing doesn't matter. This is only limited within the browser to allow malicious attacks.

It doesn't mean that your phone app can't be compromised or your desktop app couldn't be compromised. Most platforms that you're going to be building your API support CORS and it's more, it more ends up being designing which of those capabilities you're going to need to allow.

And it may just be that you want to allow CORS to be defined so that they can be changed later as you create partnerships or whether out of the gate, you know this is a public API and you're going to allow anyone in to do anything.

How does CORS work?



Just so you can really understand what's happening. **So some request comes in from another domain in a browser.**

The browser then requests access, and it does this by issuing an OPTIONS call to your server, and it's going to ask for not only what it wants to do, and in this case it wants to get at api/games, but also what kind of verb it wants to use, in this case POST.

The server replies with what you're allowed to do. Your domain is allowed and these are the methods that are allowed, and it could also be giving you the resources that are allowed if that is limited as well.

And then **the browser then issues to your server that same API request,** but it now includes the information to tell your server that this is allowed. This is where I'm coming from and this is what I'm requesting to do. **You're very rarely going to implement this yourself, but the important point is you have to realize that making the decision about CORS is part of that design process.**

AUTHENTICATION & AUTHORIZATION

Authentication is who you are, the person or the app or the server or the organization that is calling a service. It is information to determine and validate your identity. This may include credentials, like username and password, or claims, information that the server can use to identify you and make sure that you're the person you're saying you are.
This is the bouncer that's checking the ID at the door at the club. This bouncer isn't telling you what rooms you can get to in the club, only that you're allowed into the club.

Authorization, on the other hand, is what you can do based on your authentication information, based on that identity itself. These are often exemplified by rules about the rights you have. So these might be part of the claims system, this may be a role system, this may be a management system.
So authorization includes everything from saying yes I'm an admin versus a user or yes I'm a manager of these people and are allowed to. So it could be vertical or horizontal segmentation of access to a system.

In the case of authentication types for APIs :-

We're really talking about **app authentication**, and **this is where you're identifying an app for your API.** And this is something that a lot of organizations don't consider when they're in the design of an API, and that is even if you're using your API to allow individual users to log into it, you may also allow things like apps or sites or functions or containers to have access to the APIs to act as if the users. And this is really common in apps.

If you've ever opened up your bank application and put in your username and password, there has to be an authentication relationship between that app and the bank itself to know that when you pass in the user's credentials that not only is the user allowed into the API, but so are you. **This is authenticating the developer themselves.** This is **typically handled through an AppID and a key,** but could be done with certificates and some other methods. You want to consider that if your API needs to then have third party application support, you're going to need to support app authentication.

In addition, there's going to be **user authentication,** and this is much more traditionally what we think of as authentication. And **this is identifying the user, usually through credentials, claims, third party authentication like OAuth.** These are the typical ways we think of doing authentication.

AUTHENTICATION TYPES
(Primarily 4, can be many more…)

COOKIES

Using cookies for our authentication is a bit tricky. **The problem is they're easy, often websites that are using cookies for their own authentication means would like the API to handle them as well.** This is easiest and actually pretty common out there, but **it is subject to request forgery pretty easily. It's not super secure.**

If you're building your bowling team's authentication, it's probably going to be fine. If you're building your bank's authentication, this is a big danger sign. The amount of risk is really high. And so whether you use cookies or not, I tend to tell people to not bother, just because the security ramifications are pretty high if you're wrong. So it really depends on what kind of security needs you have.

BASIC AUTH

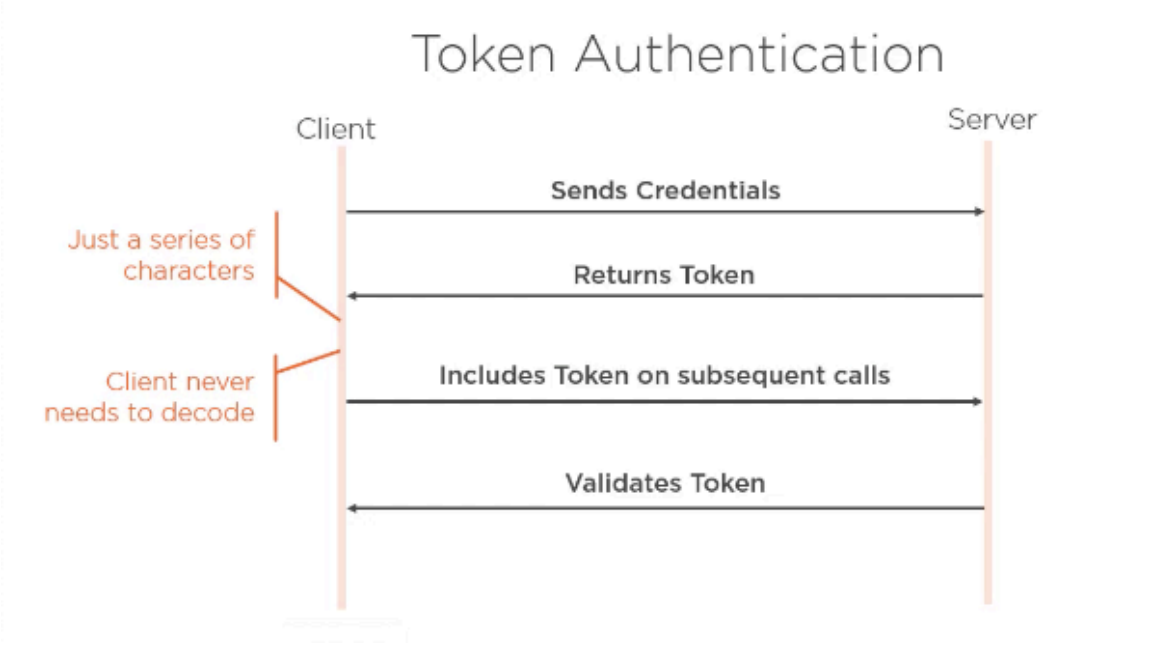
This is really easy to implement because basic auth allows you to pass in information, in the query string or in headers, with the credentials to validate on the server. But it's super not secure. [#RISKY]

Even in SSL, you can leak username and passwords to the actual users themselves. I don't suggest basic auth because it does require SSL across the wire, but you're passing those credentials on each and every request, and the more you pass a credential, the more chance of someone intercepting it and getting it. Now again, with SSL this goes way down, but it still represents risk.

So basic auth ends up being risky for a couple of reasons. One, it sends the credentials on every request, which makes the surface area for these attacks higher. The more often you're transporting over the internet, usually a set of credentials, the more chance that someone's going to be able to grab them. Remember the people grabbing them aren't just the users, they can be man-in-the-middle attacks or people attacking them on the server with malware. So basic auth is possible. You can make it pretty secure, but it's still risky.

TOKEN BASED AUTHENTICATION

This is most commonly used and has a mix of both being secure and simple. There's a standard out there for these tokens and so there's usually middleware that will support creating and validating these tokens on most platforms. These tend to be more secure than cookies, because they usually expire much faster, so the chance of getting that token and reusing it, the window is usually much, much smaller.



When a client is calling the server to use an API, they usually start with an API call that accepts credentials, again, only through over SSL, otherwise we're just as risky as basic authentication, and that the server then returns a token, assuming those credentials were good.

And this token is just a series of characters typically. The client never needs to decode this set of characters, it just needs to pass them along on the other requests it wants to make. The server validates the token and returns whatever the client wanted.

Now the bottom half of this chart repeats until that token expires.

And so the idea here is that you're making multiple requests, not on every call, but only when you start to get that initial token. You hold on to and continue to send that token until it times out.

JSON WEB TOKENS

The most common of these token systems is the JSON web token. **It's an industry standard. It's self-contained and can be small and complete with all the information needed to validate.**

The token itself contains information about the user who created the JWT and usually that means the end user. Includes claims for the kind of rights it has. Has a validation signature so that the server can look at it and go, yes no one has edited the data after I've issued this token. And it can contain an X number of other information.

Now what's interesting about JSON web tokens is **they're not encrypted.** The information in the token itself isn't protected against people getting at it, but what the web token has is this

validation signature so that it knows that once a server issues it, it can then also validate that it's still the same one it passed to it.

And so in this way, the JSON web tokens are simply a package for a small amount of data that includes information about the user. Now instead of having something sticky or keeping state on the server, using JSON web tokens allows those tokens to be used in large scale situations where you may not be going to the same server to look up the same user. These web tokens are normally self-contained with all the information that is needed in order to validate a user has rights to some API. This is the most common thing I suggest for APIs, though there are occasions when you want to use third party authentication instead.

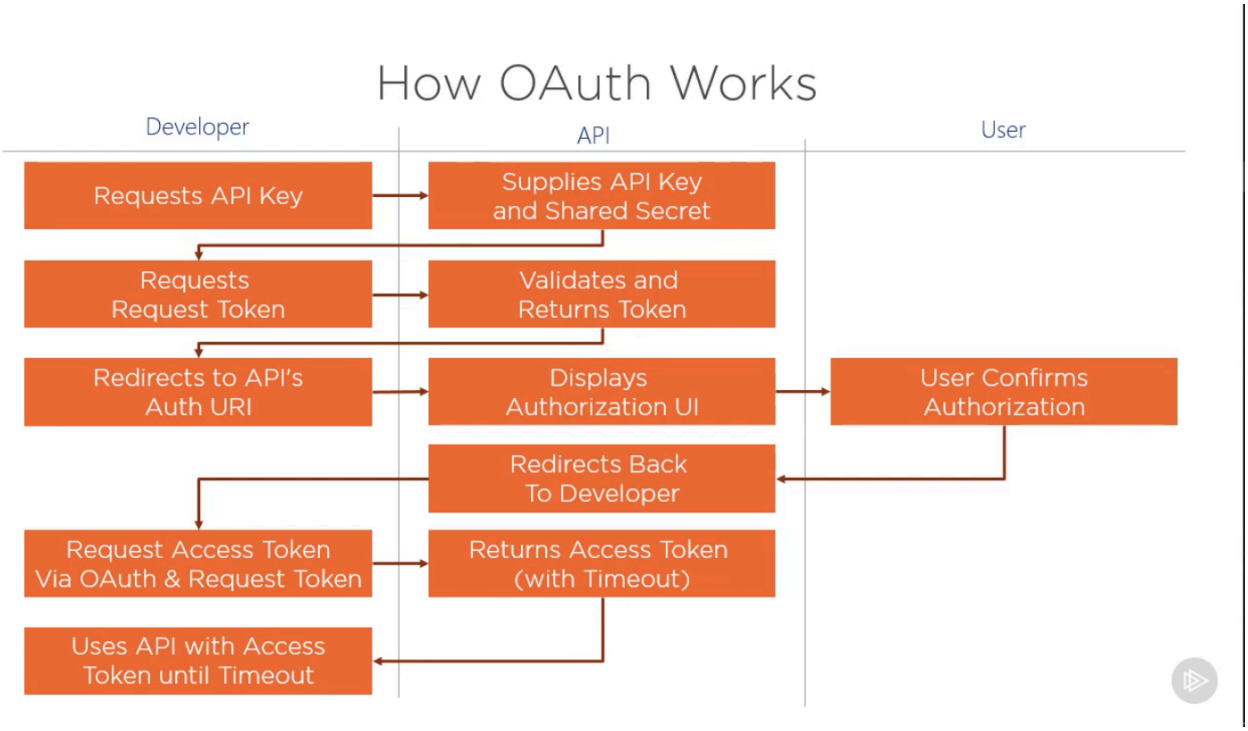
OAUTH

OAuth has a couple of standards out there, but it's used to allow trusted third parties to identify users. In this way the app that is using OAuth never gets the credentials. So you're not leaking your username and password, or whatever credentials you're going to use, to another party. This allows that if you're using some one-off website to do something interesting that you can say, hey I have an account over at Google or Microsoft, let me log into it and then this third-party app can trust Microsoft to have validated that as a real user.

This lowers the surface area in many ways. **But it really is for certain use cases where you're dealing with third parties and wanting your users to trust that the API isn't going to have a massive data leak, like we see virtually weekly these days. So in that way, OAuth never receives the credentials, the user authenticates with that third party, and then uses another kind of token to confirm the identity. It's safer for you, the developer, because you don't have to have the responsibility of holding onto more user information.**

This is really the best way to handle situations where you want your API to be used by third party developers for them to use them in apps or games or whatever else you're using for your API.

How does OAUTH works?



The developer decides it wants to allow OAuth with your API and so it requests an API key from you. Might do this through the website, might do this through an API, but there has to be some sort of handshake that says this is how I trust you as a developer to access my API. This isn't about the users themselves.

With that API key and shared secret in place, the developer requests a token from the API, looks at the information it's given, and returns that token. That token then indicates that we need to redirect the user to the actual partner that's going to do the authentication.

This authorization UI is displayed to the user, though it's not hosted by your API, it's hosted by whatever third party you're going to allow Google, Microsoft, GitHub, etc. and the user confirms that authorization, yes I'm going to allow this third party to do this. It redirects back to the developer's website, which then requests an access token, and the API supplies it much like the way the JWT works.

And then finally, the developer uses the API with the access token until the timeout, just like JWT, but in this case it's using the API as the user, not as the developer. This is a key difference in how OAuth works and why it's so important to use OAuth if you're dealing with third party developers and users, because you don't want to make a requirement that the user passes usernames and passwords to the developers in order to use your API. This tends to be more niche than a lot of developers think it will be.

If you're building an API to service your website and your app, OAuth doesn't matter because you would be the trusted source of the credentials anyway.

So what this comes down to is OAuth is crucially important if you need this level of security, if you don't want to have to deal with holding onto credentials or you want to have third-party developers allow that. Never ever try to implement this by hand because, as I've shown you, it is complex. But OAuth should be considered when you're looking at your design requirements to see what really is going to be required.

Summary:

Even though a lot of these security implications for your API feel like they should be sort of above the API, they should always be considered while you're designing your API because you're a good developer who takes API design seriously, so you should take it seriously as a whole, not assuming that just what the URL looks, just what the payload looks like is what is part of your job.

Making sure that the wrong people can't get into the API to get those payloads is just as important and just as much your job. It doesn't mean you need to be a security expert, but you need to consider the level of security you're going to need when designing your API.