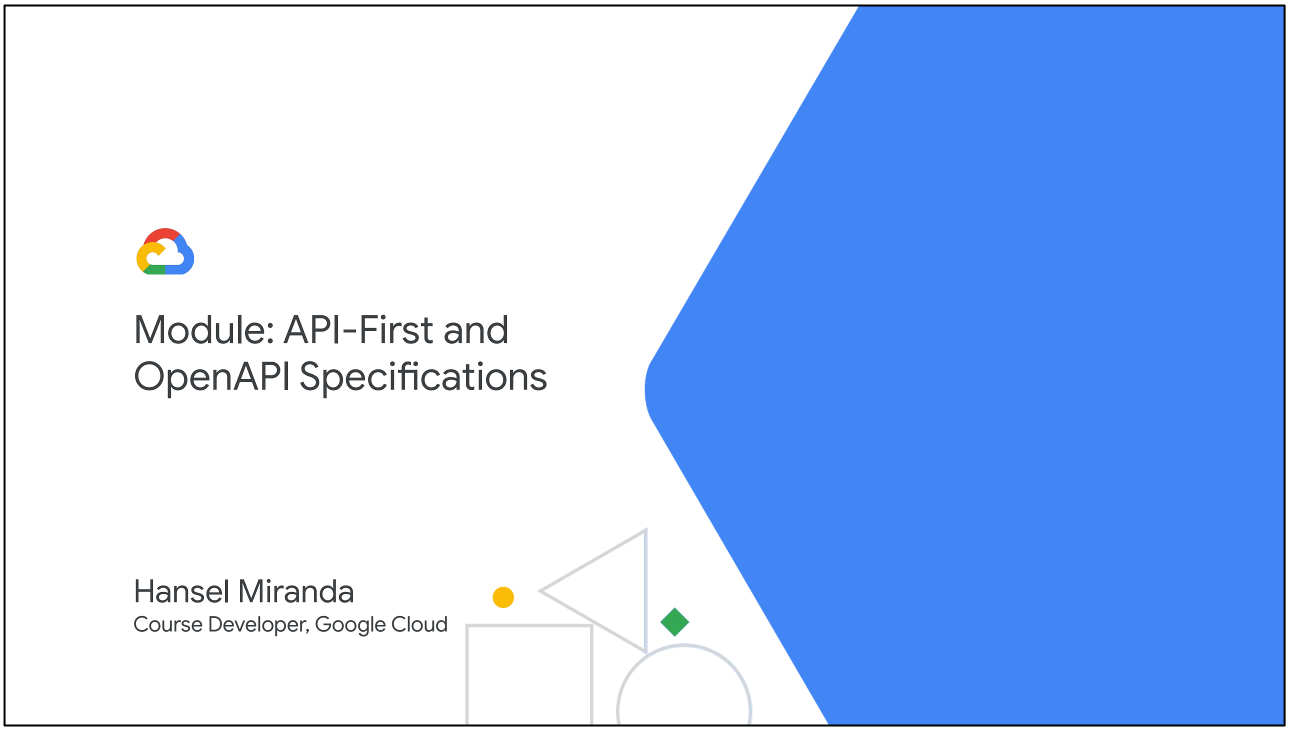


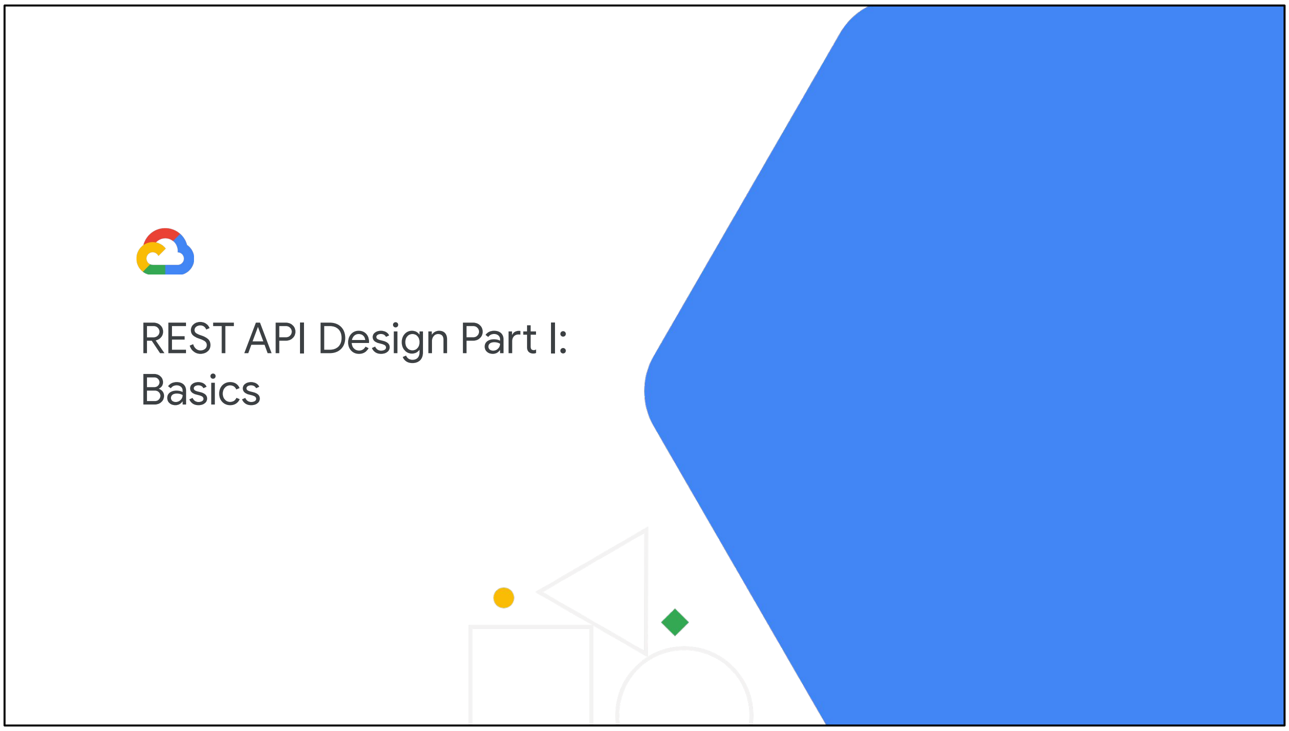
Module: API-First and OpenAPI Specifications

Hansel Miranda
Course Developer, Google Cloud

In this module, you'll learn the basics of REST APIs and how we design them.

This module will teach you about API-first development, which recommends that you design APIs with the app developer in mind.

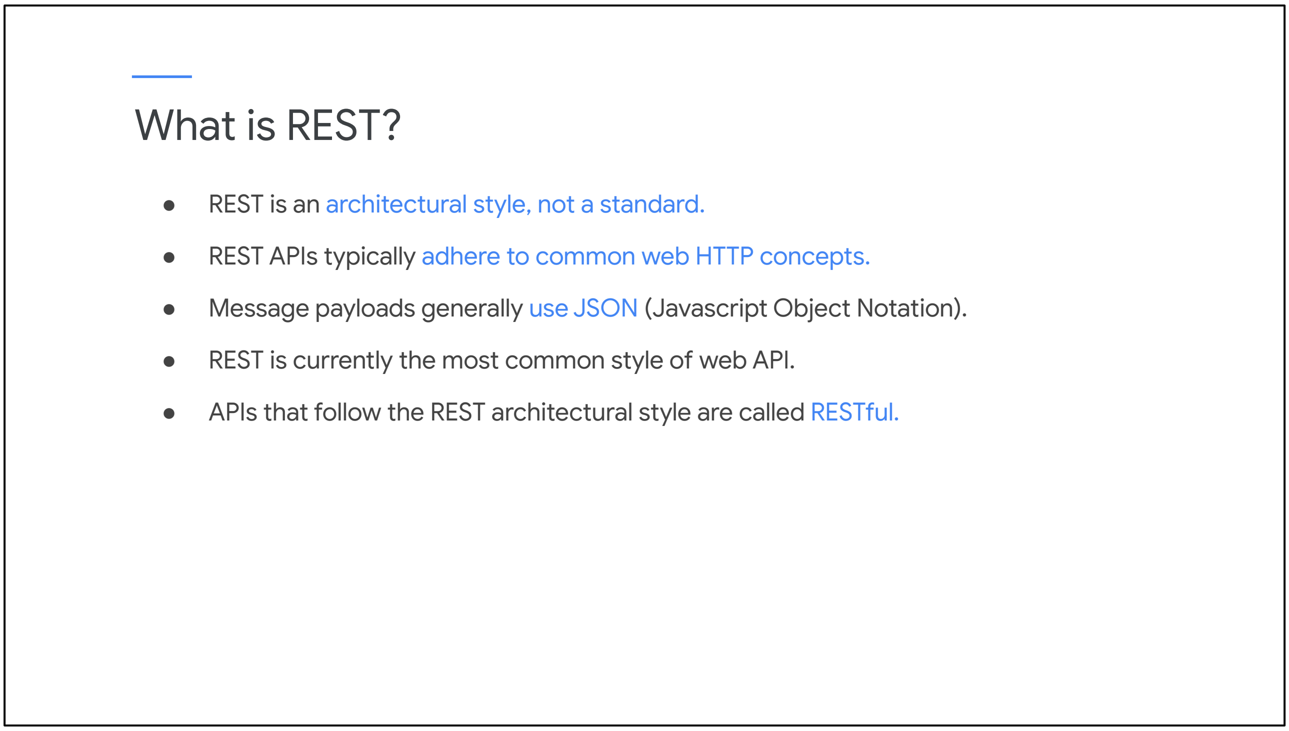
You'll also learn about OpenAPI specifications, which are used to document APIs and provide live documentation in an Apigee developer portal.



REST API Design Part I: Basics

This lecture is the first of three on REST API Design.

We'll discuss the primary features of REST APIs, and you'll learn how to leverage HTTP concepts for those APIs.



What is REST?

- REST is an architectural style, not a standard.
- REST APIs typically adhere to common web HTTP concepts.
- Message payloads generally use JSON (Javascript Object Notation).
- REST is currently the most common style of web API.
- APIs that follow the REST architectural style are called RESTful.

Before we get into the basics of designing REST APIs, let's explore what we mean by the term "REST." REST was defined in Roy Fielding's 2000 doctoral dissertation at the University of California, Irvine.

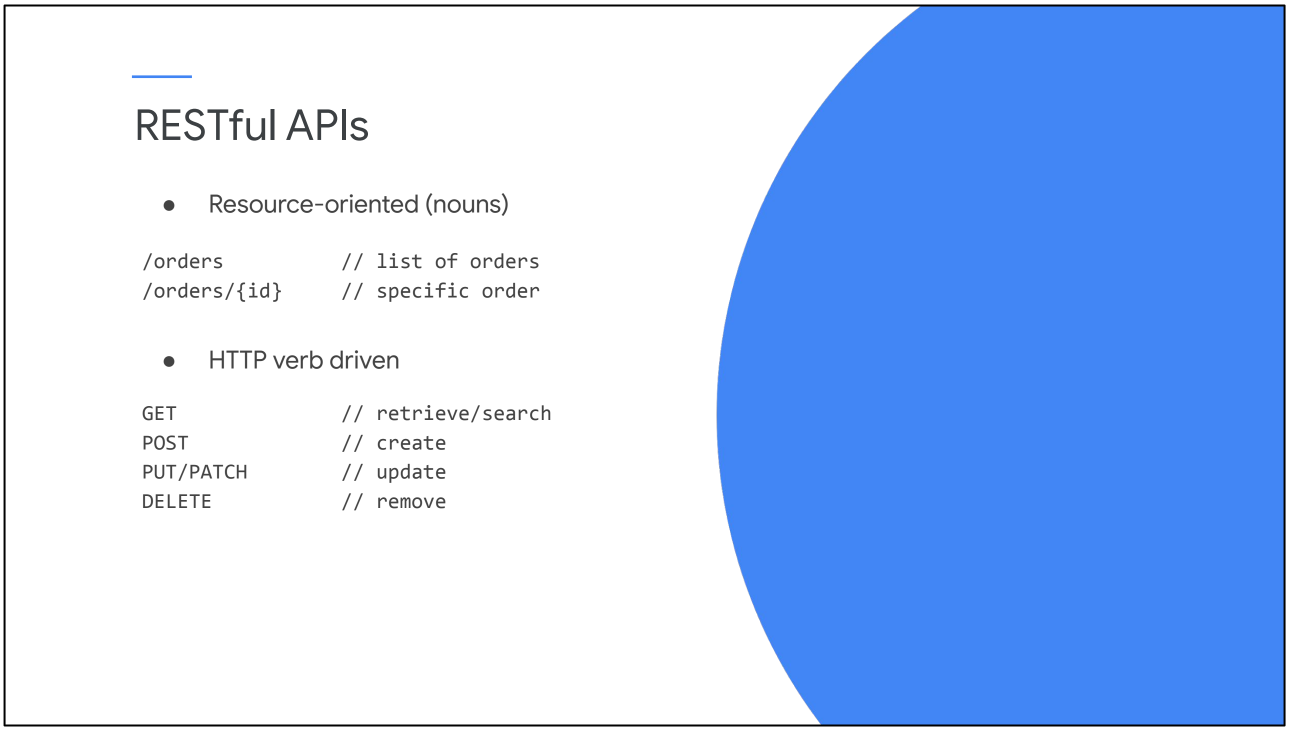
At this time, SOAP, or Simple Object Access Protocol, was a popular way to implement APIs. SOAP, however, is not very simple. In order to make an API call using the SOAP protocol, a developer needs to craft a complex XML payload by referencing an even more complex definition document called a WSDL, which stands for Web Services Description Language.

Fielding and his colleagues designed the REST architectural style while version 1.1 of HTTP was being designed. They saw an opportunity to use HTTP concepts to create a simpler and more familiar pattern for APIs. REST APIs leverage common web HTTP concepts like URLs, verbs, and status codes.

Message payloads for REST APIs are generally specified using simple JavaScript Object Notation, or JSON. The JSON payload of a well-written REST API is simpler and more human-readable than the complex XML used for SOAP services.

The use of common web patterns and simple payloads makes REST APIs easy for app developers to learn and use. Due to its simplicity and ease of use, REST has become the predominant style of web API today.

When an API is designed using the REST style, we call it a RESTful API.



RESTful APIs

- Resource-oriented (nouns)

```
/orders          // list of orders
/orders/{id}     // specific order
```

- HTTP verb driven

```
GET              // retrieve/search
POST             // create
PUT/PATCH        // update
DELETE           // remove
```

RESTful APIs are resource-oriented, focusing on the resources being acted upon, instead of focusing on a list of operations or actions.

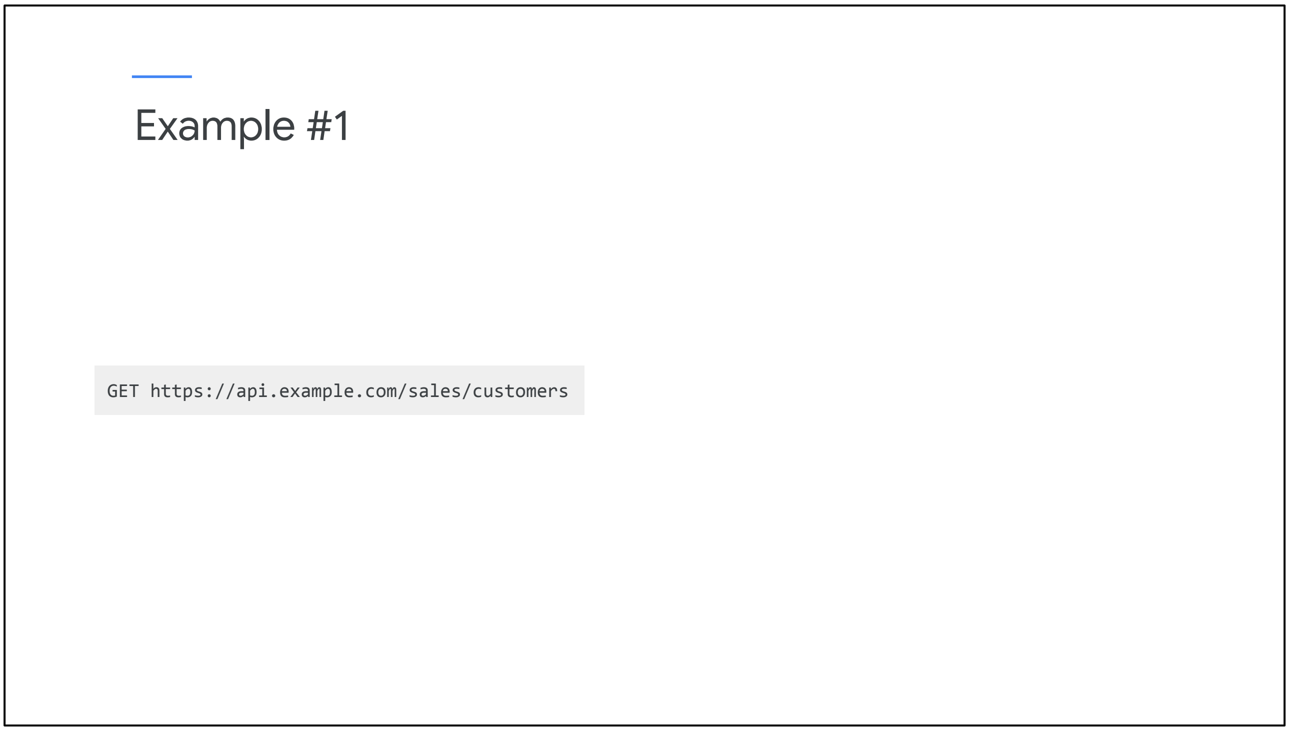
Resources are specified in API URLs by using resource names and IDs. For example, to access a list of orders, you would make a request to "/orders." To access order 1234, you would make a request to "/orders/1234."

The request's HTTP verb, or method, specifies the type of operation that is being performed on the resource or list of resources.

Normal browser requests to retrieve web pages use the verb GET. A GET request for a RESTful API therefore retrieves details about the resource or resources specified in the URL.

POST is used to create a new entity. POSTing to "/orders" would typically create a new order with an auto-generated ID.

PUT or PATCH to "/orders/id" is used to update an existing order, and DELETE on "/orders/id" would remove the order.

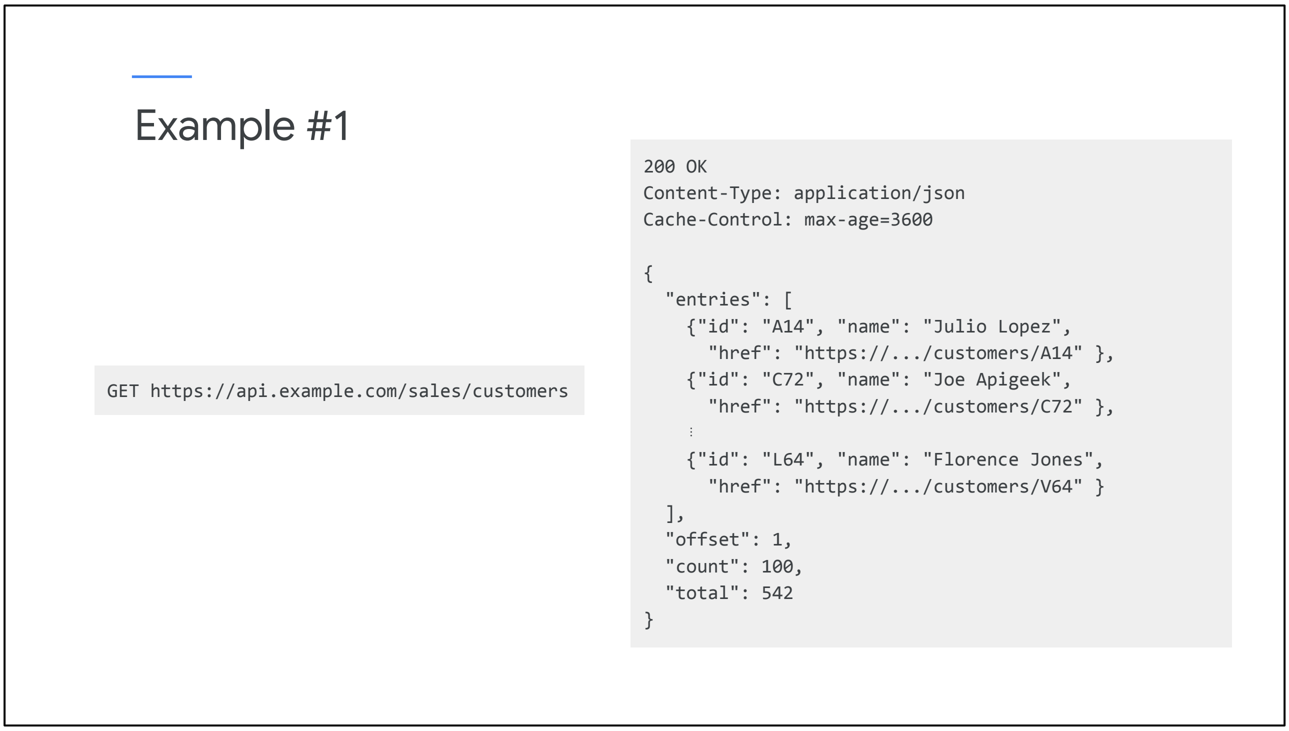


Example #1

```
GET https://api.example.com/sales/customers
```

See if you can figure out what this API request means.

What do you think this request is trying to achieve?


Example #1

```
GET https://api.example.com/sales/customers
```

```
200 OK
Content-Type: application/json
Cache-Control: max-age=3600

{
  "entries": [
    {"id": "A14", "name": "Julio Lopez",
      "href": "https://.../customers/A14" },
    {"id": "C72", "name": "Joe Apigeek",
      "href": "https://.../customers/C72" },
    ⋮
    {"id": "L64", "name": "Florence Jones",
      "href": "https://.../customers/V64" }
  ],
  "offset": 1,
  "count": 100,
  "total": 542
}
```

If you said that this URL retrieves a list of customers, you'd be correct.

Look at the beginning of the request URL. This API request is using the domain name api.example.com, and the beginning of the URL path is "/sales." For a REST API, the first part of the URL specifies the particular API that is being called. So this is example.com's sales API.

After "/sales," we see "/customers." This follows the resource pattern we saw earlier. "/customers" represents a list of customers. We are using the GET verb, so this request is fetching a list of customers.

On the right is a typical response for the GET customers call.

200 is the response status code, and OK is the reason phrase. The status code indicates the success or failure of the request, as well as the type of failure, if the request failed. Since the API returned 200 OK, the API call succeeded.

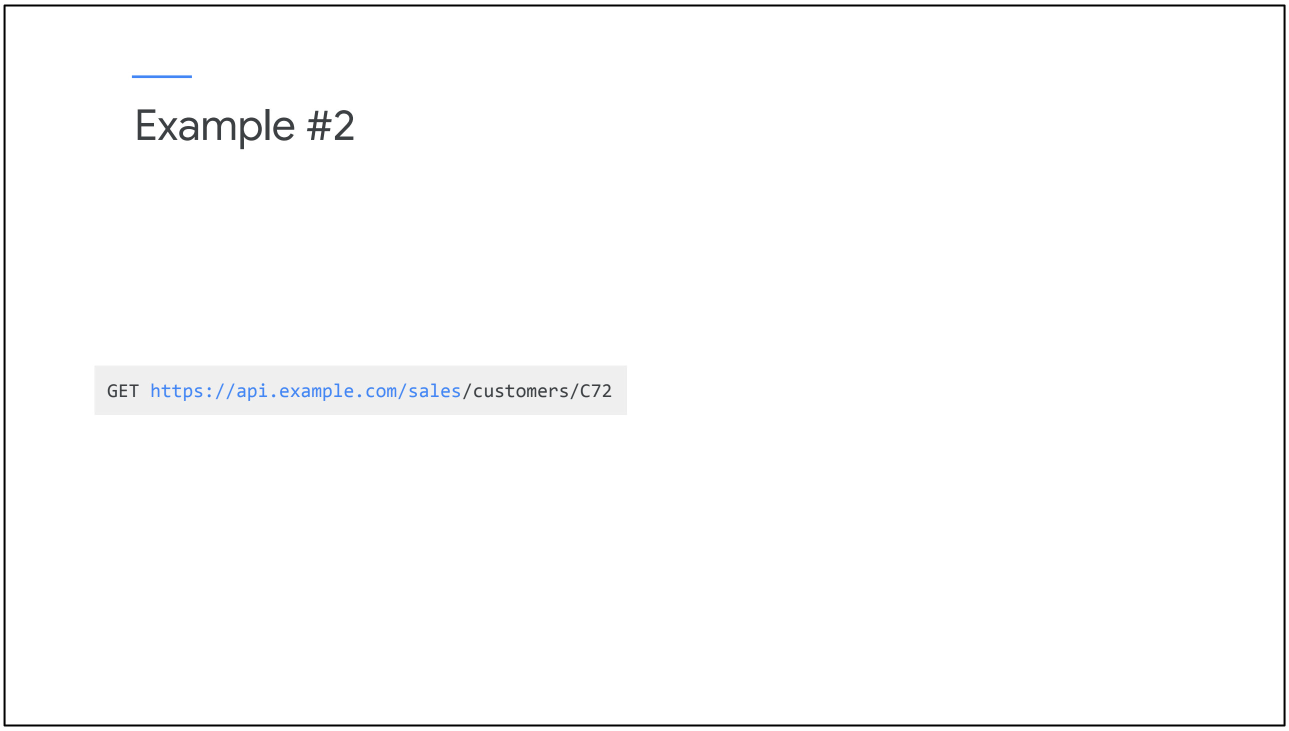
The next two lines are headers. The Content-Type header specifies the type of data returned in the payload: JSON, in this case. The Cache-Control header specifies how long the requestor can cache and use this response instead of having to make the same request again.

After the headers is the response payload. The JSON response is showing a list of customers, with an ID, name, and reference for each. The href reference is a URL that can be used to retrieve more details about that customer.

We also see additional metadata about the response: the offset of the first customer retrieved and the number of customers in the payload.

According to the count, 100 customers were returned. The response is truncated to show only a few entries. The offset to the first customer is 1, so this response is listing the first 100 customers. The total number of customers, according to the metadata, is 542.

Remember that REST is an architectural style, not a standard. The payloads of REST responses do not have a required format, but you should try to keep responses consistent for all of your APIs, and especially within an API.



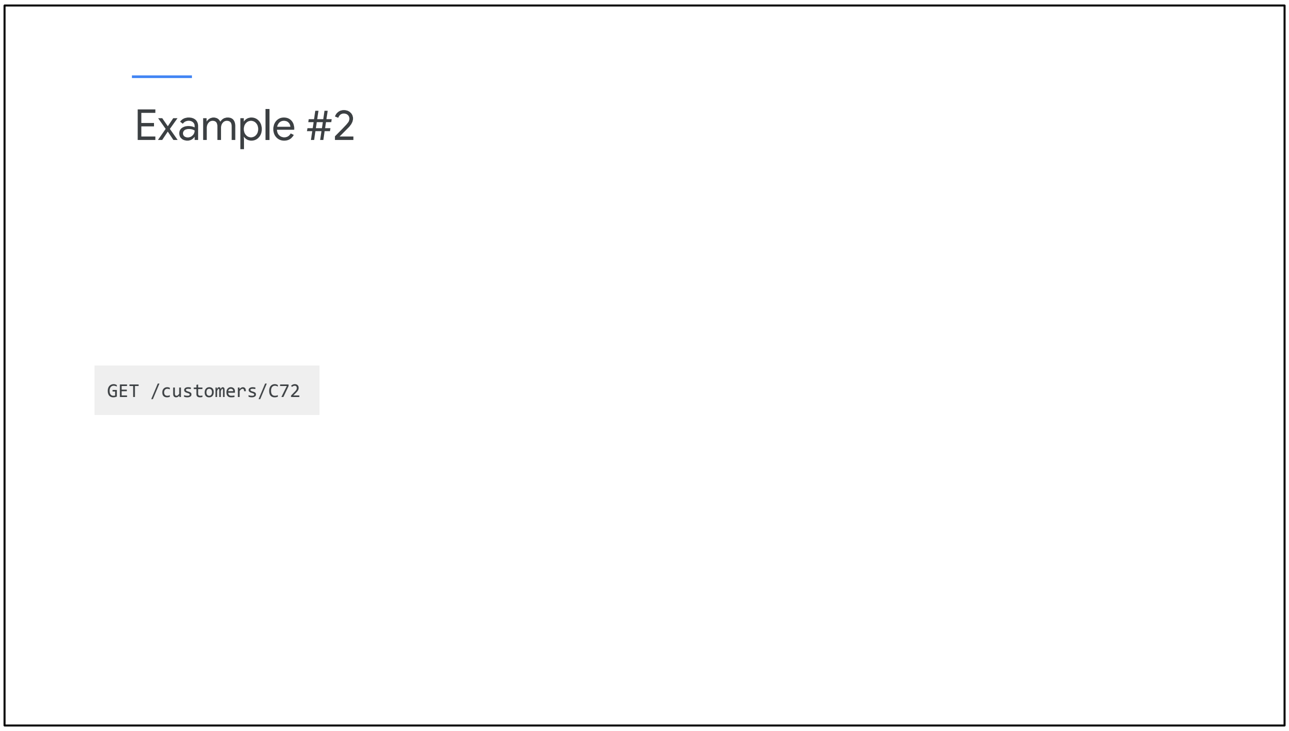
Example #2

```
GET https://api.example.com/sales/customers/C72
```

OK, here's another one.

We know from the previous example that https://api.example.com/sales is the root URL of example.com's sales API.

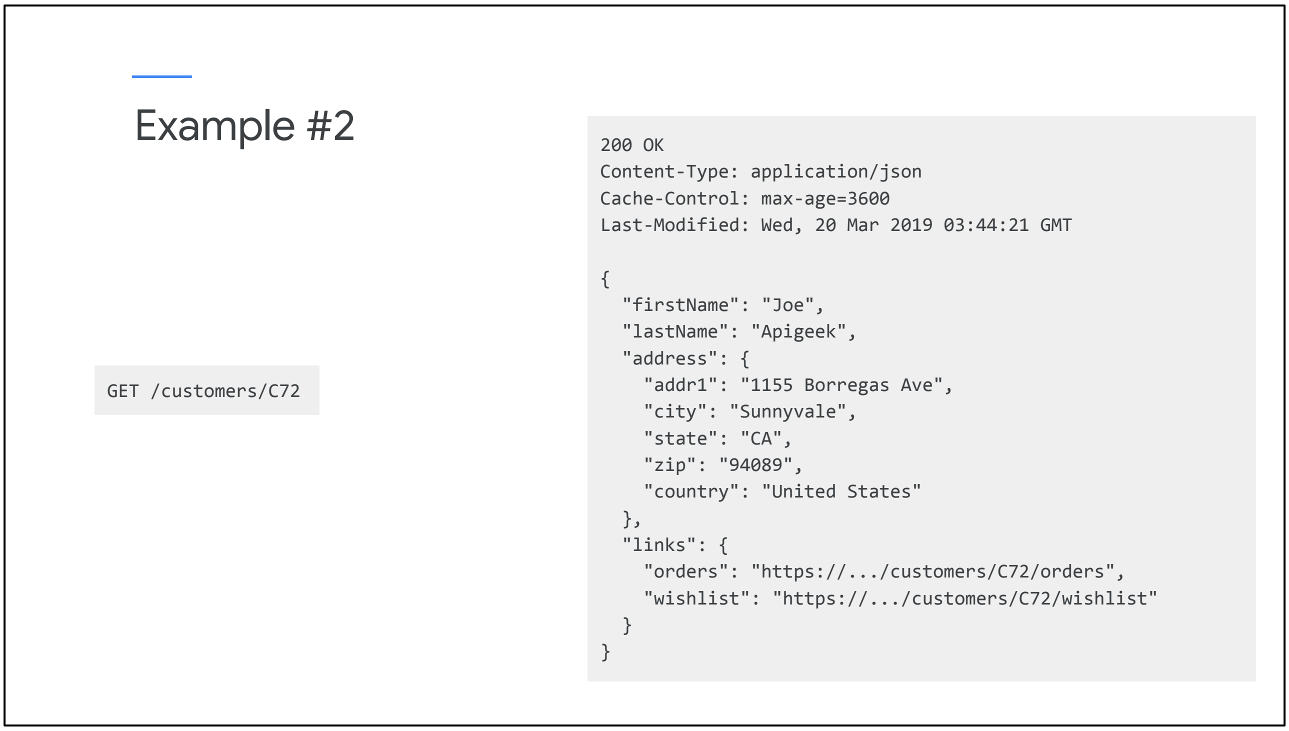
When we are specifying calls in the context of an API, we often shorten this by removing the root URL. We would call this a GET on "/customers/C72."



Example #2

```
GET /customers/C72
```

What would GET on "/customers/C72" signify?



Example #2

```
GET /customers/C72
```

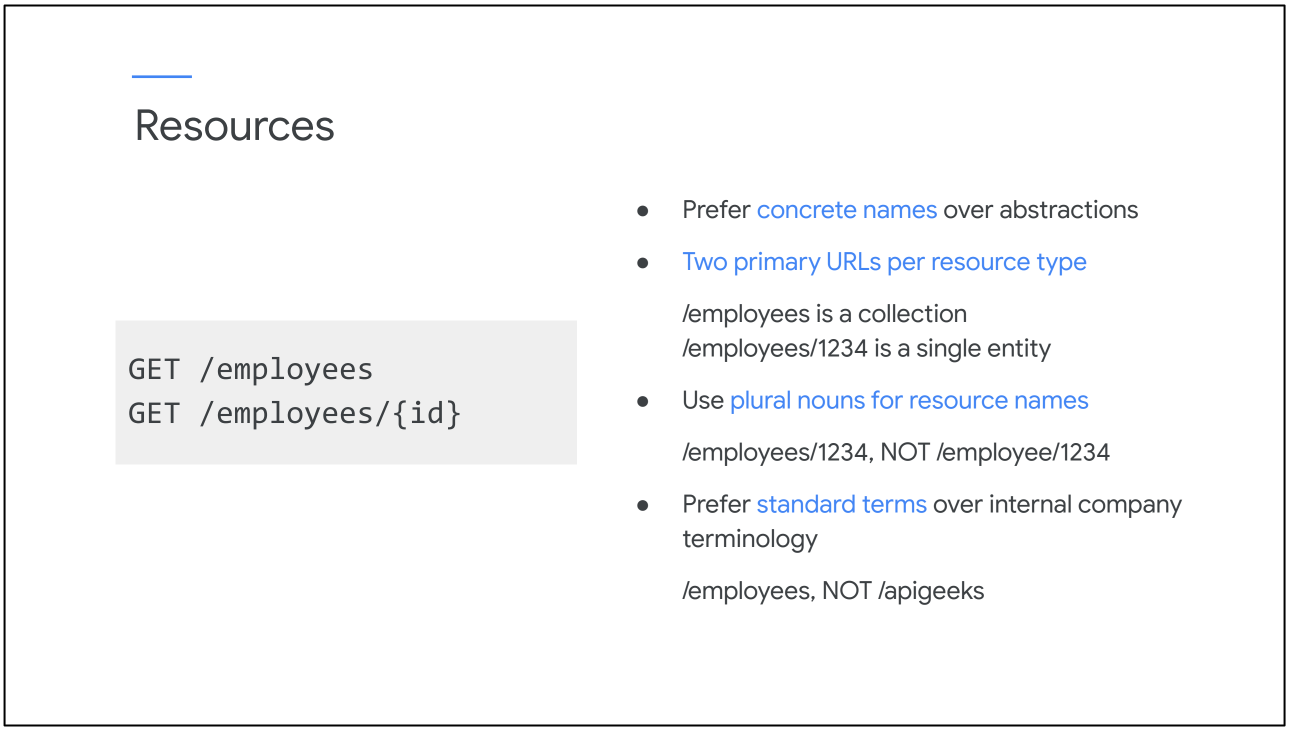
```
200 OK
Content-Type: application/json
Cache-Control: max-age=3600
Last-Modified: Wed, 20 Mar 2019 03:44:21 GMT

{
  "firstName": "Joe",
  "lastName": "Apigeek",
  "address": {
    "addr1": "1155 Borregas Ave",
    "city": "Sunnyvale",
    "state": "CA",
    "zip": "94089",
    "country": "United States"
  },
  "links": {
    "orders": "https://.../customers/C72/orders",
    "wishlist": "https://.../customers/C72/wishlist"
  }
}
```

This retrieves information for a specific customer, in this case Joe Apigeek, who has an ID of C72.

In this example, we retrieve much more than just the name, ID, and a reference, since we are requesting the details of a single customer instead of up to 100 of them. This response also includes links, which allow us to easily retrieve the orders or wishlist of Joe Apigeek.

When you design your own APIs, think about how your app developers would want to consume your APIs, and what data would be useful to them. Remember, your app developers are your customers.



Resources

```
GET /employees
GET /employees/{id}
```

- Prefer concrete names over abstractions
- Two primary URLs per resource type

  /employees is a collection
  /employees/1234 is a single entity
- Use plural nouns for resource names

  /employees/1234, NOT /employee/1234
- Prefer standard terms over internal company terminology

  /employees, NOT /apigeeks

A core part of designing your APIs is identifying your resources.

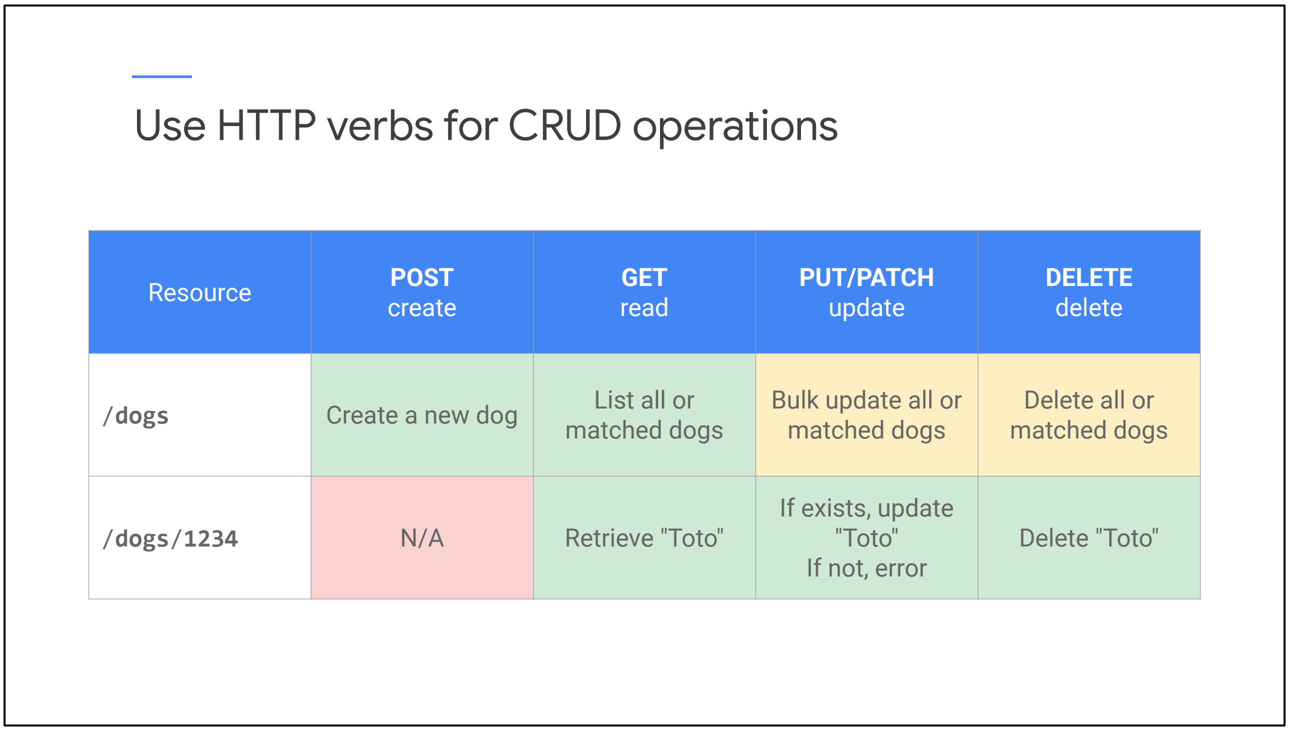
Resources typically use concrete names instead of abstractions. In an employee API, your resources might include employees, buildings, and reporting relationships.

Each resource type typically has two primary URLs: /employees would be a collection of employees, and /employees/1234 would be the specific employee with the unique ID 1234.

We use plural nouns for resource names in the URL. Use the plural noun for the collection name, "/employees," as well as for a specific employee, "/employees/1234."

Remember also that you are designing your resources to be understood by app developers, who may not have knowledge of internal names and concepts used within your company. You should always use terminology that makes sense to your app developers and is meaningful to to them.

For example, employees of the Apigee team at Google are sometimes called "Apigeeks." This term could likely be confusing to most app developers, including app developers within Google. It would make more sense to name the resource "employees."

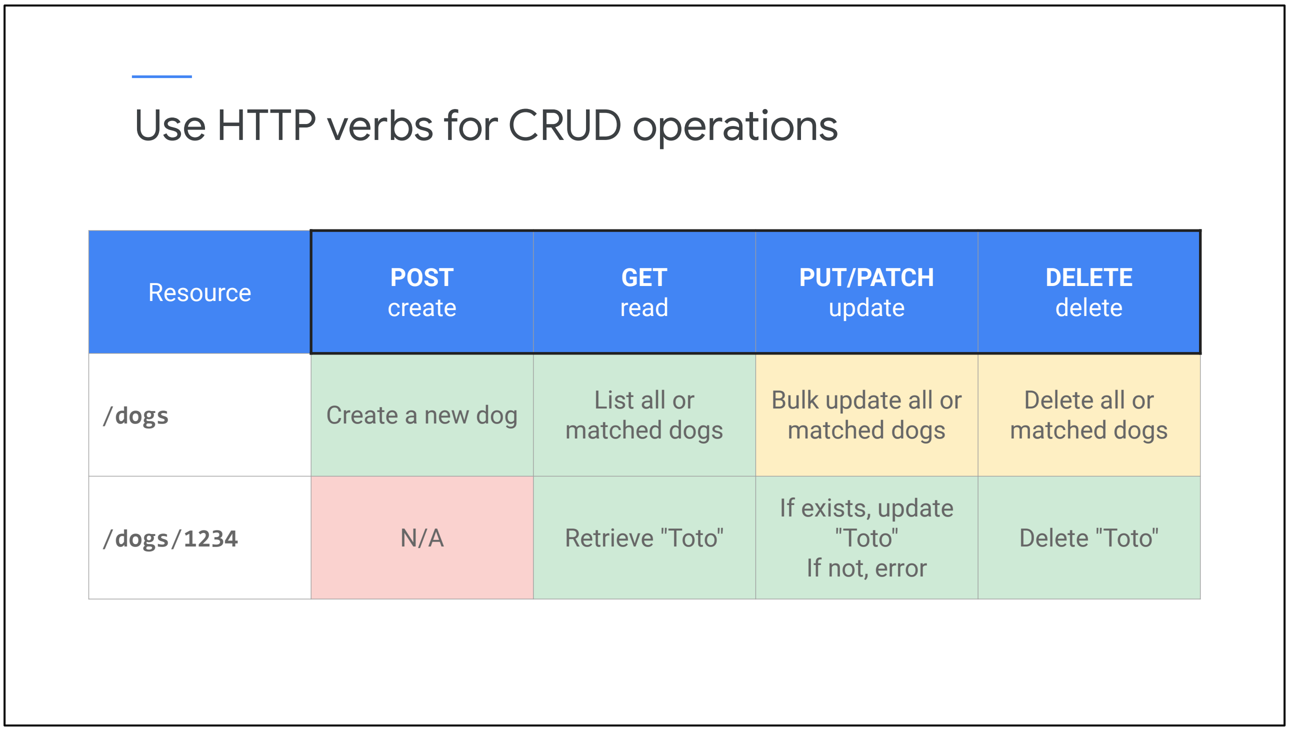


Use HTTP verbs for CRUD operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

As we saw before, HTTP verbs are used to specify the type of operation being performed on a resource.

Using dogs as our resource, let's discuss how we should map HTTP verbs for the required operations.



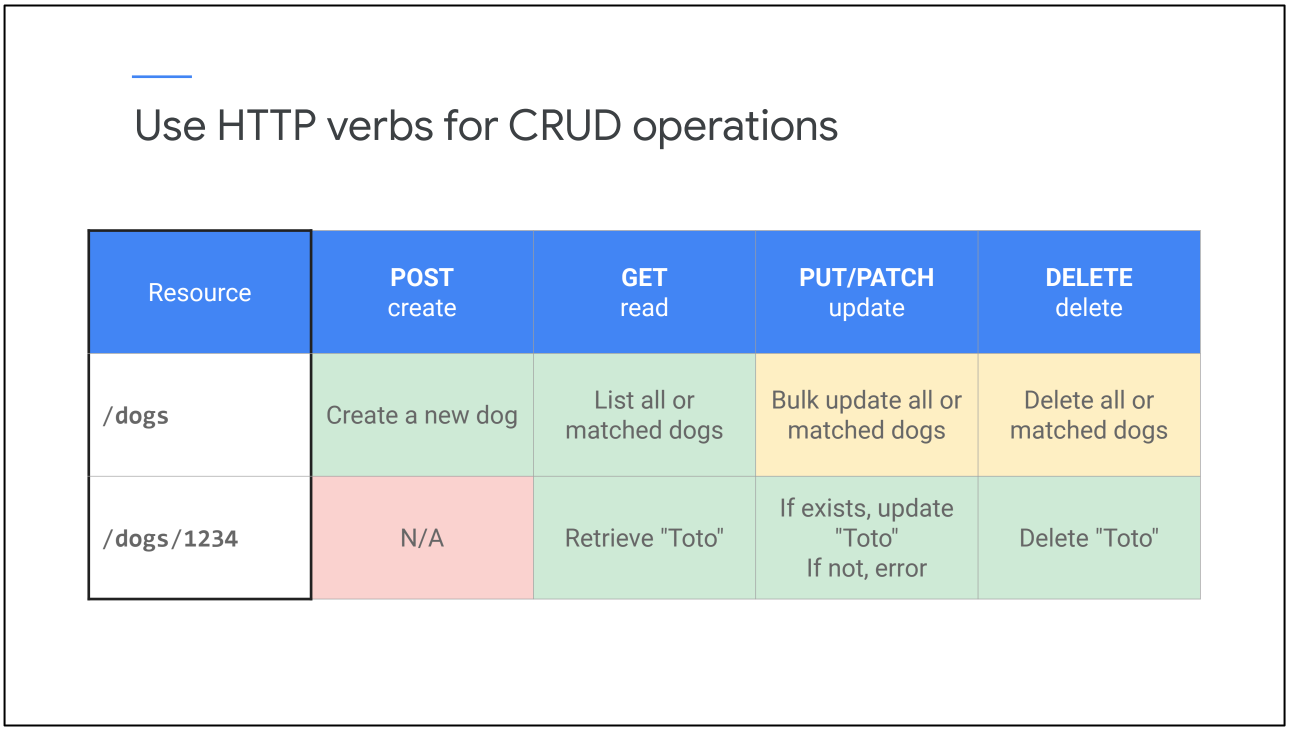
Use HTTP verbs for CRUD operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

We can map these HTTP verbs to the four basic functions of persistent storage: Create, Read, Update and Delete.

We typically refer to these as "**CRUD**" operations.

CRUD is an acronym for Create, Read, Update, and Delete.

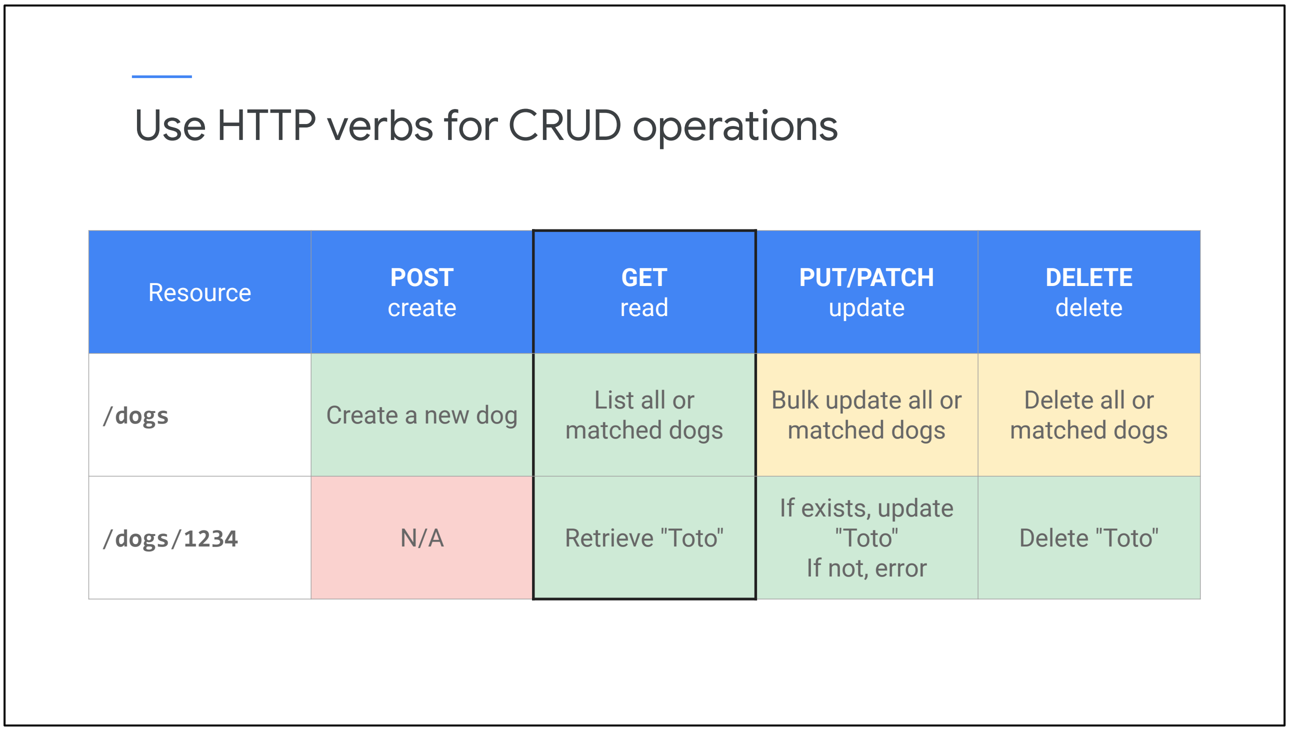


Use HTTP verbs for CRUD operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

Remember that we have two primary URLs for our resources: "/dogs" refers to a collection of dogs, and "/dogs/1234" is the specific dog with the id 1234.

Let's see what CRUD operations can be performed on dogs using these HTTP verbs.



Use HTTP verbs for CRUD operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

GET is the verb used to retrieve information about a collection of resources or a single resource. When you retrieve a web page in your browser, you are using the GET verb.

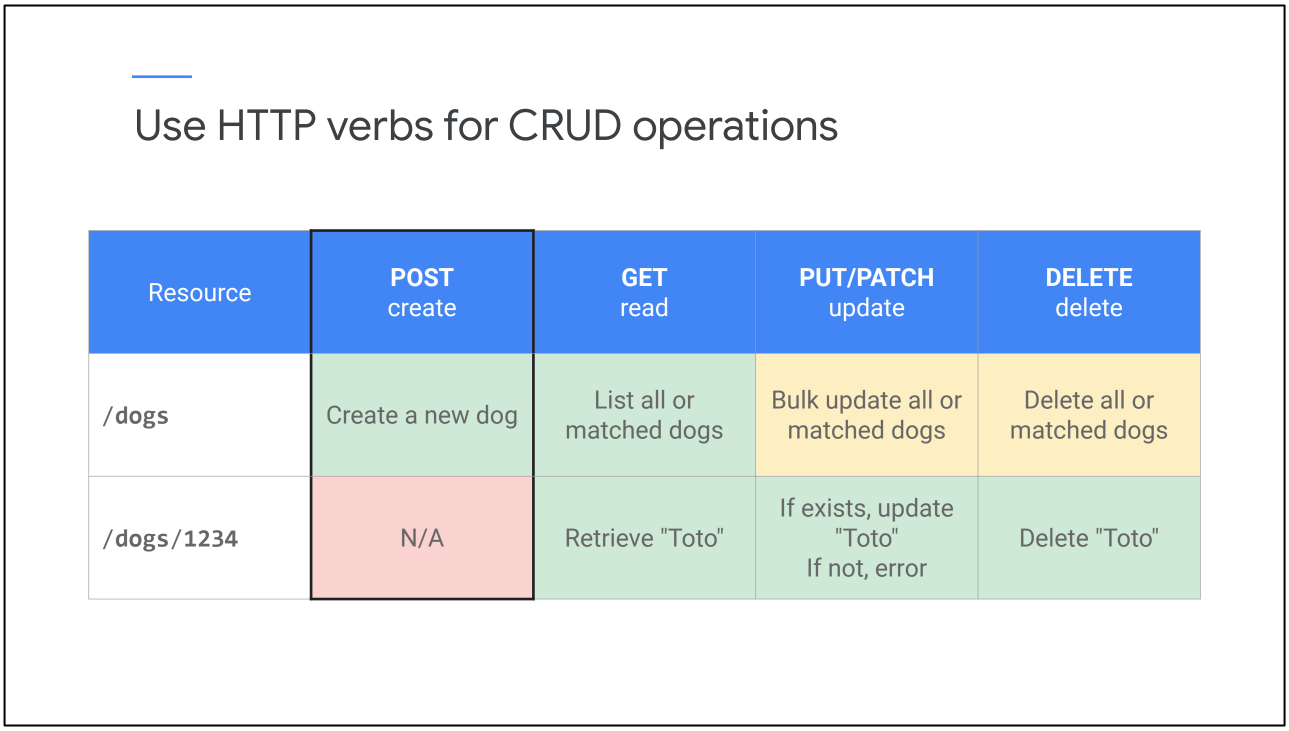
A GET should not update the state of the resource at all.

An operation that does not update the state is known as a **safe** method.

Unless another API call has modified the state of your resource between calls, repeating the same GET call will always produce the same result.

We call this being "**idempotent**."

HTTP specifies that GET should be safe and idempotent, so make sure your API adheres to these rules.



Use HTTP verbs for CRUD operations

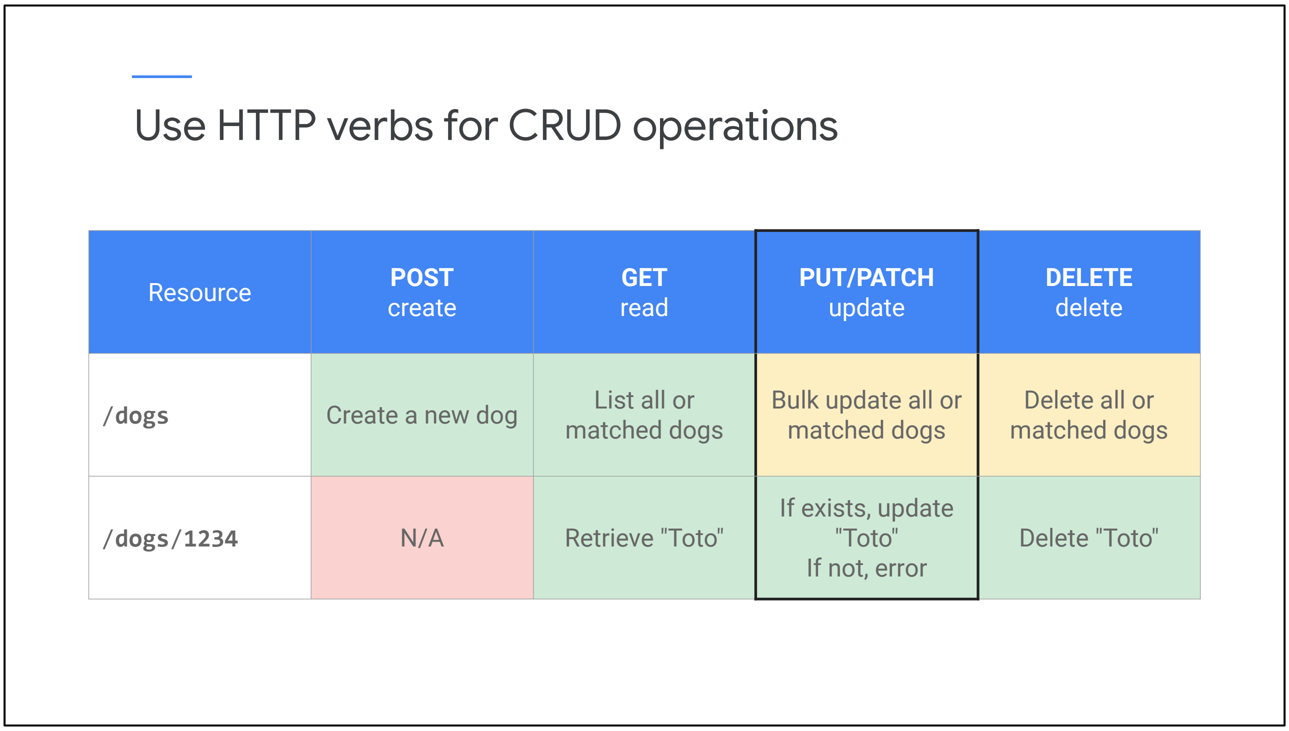
Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

POST on "/dogs" is used to create a new dog. No id is provided in the URL; the id for the new dog should be automatically generated by the API.

The POST method modifies the state of resources, creating a new resource in the dogs collection. Therefore, POST is not a safe method.

Repeating the same POST call will result in multiple dogs being created, so POST is also not idempotent.

POSTing to a specific dog by id does not have a specific CRUD meaning.



Use HTTP verbs for CRUD operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

For update, we have two methods listed: **PUT** and **PATCH**. We will focus on the differences between PUT and PATCH later.

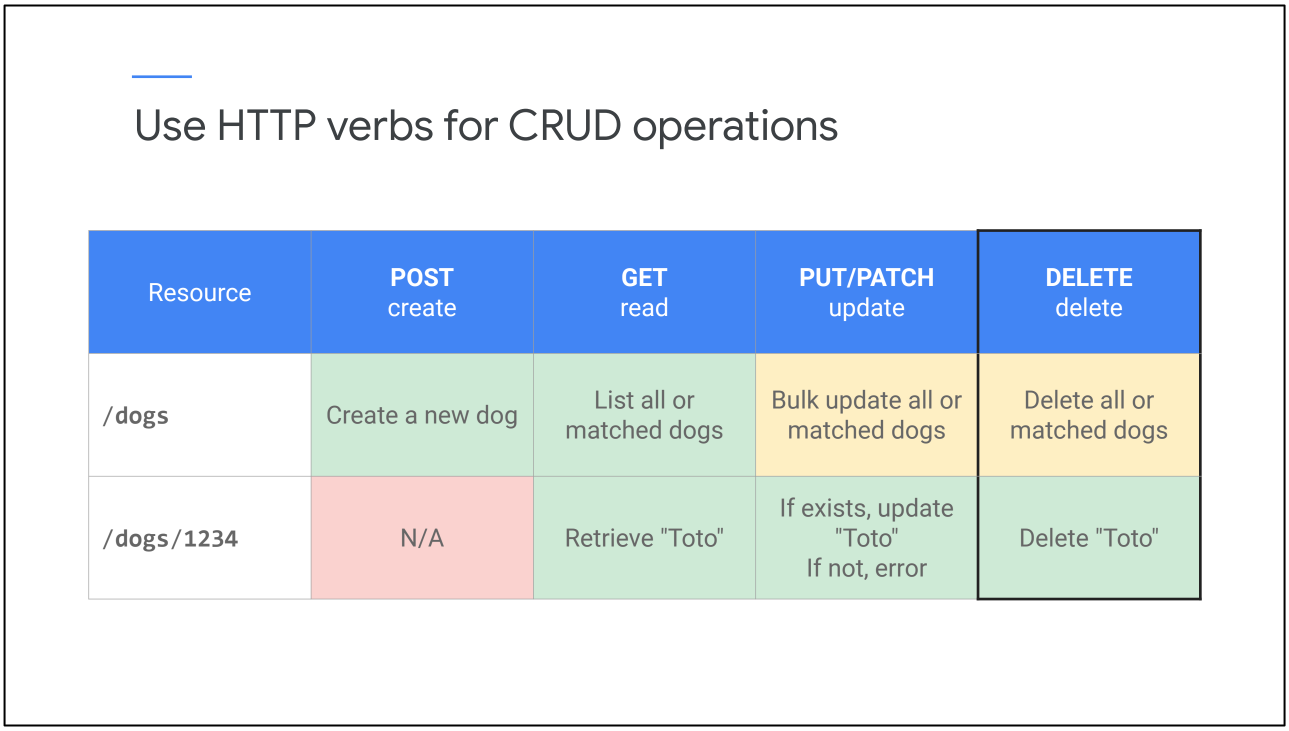
"PUT /dogs/1234" would update Toto, the dog with id 1234.

If appropriate for your API, you can have PUT create a new resource with the specified id if one doesn't already exist. This is somewhat rare: typically this would only be used when the id of a resource is a unique number or string, like a product ID.

"PUT /dogs" would bulk update all dogs, or matching dogs if your request included a search or filter. Do not allow this operation unless it is necessary, because it might be too easy to accidentally bulk update lots of dogs.

PUT is not a safe method, because it is meant to update resources.

PUT, however, is specified by HTTP as an idempotent method. A second identical PUT call should therefore leave the resource or resources in the same state.



Use HTTP verbs for CRUD operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

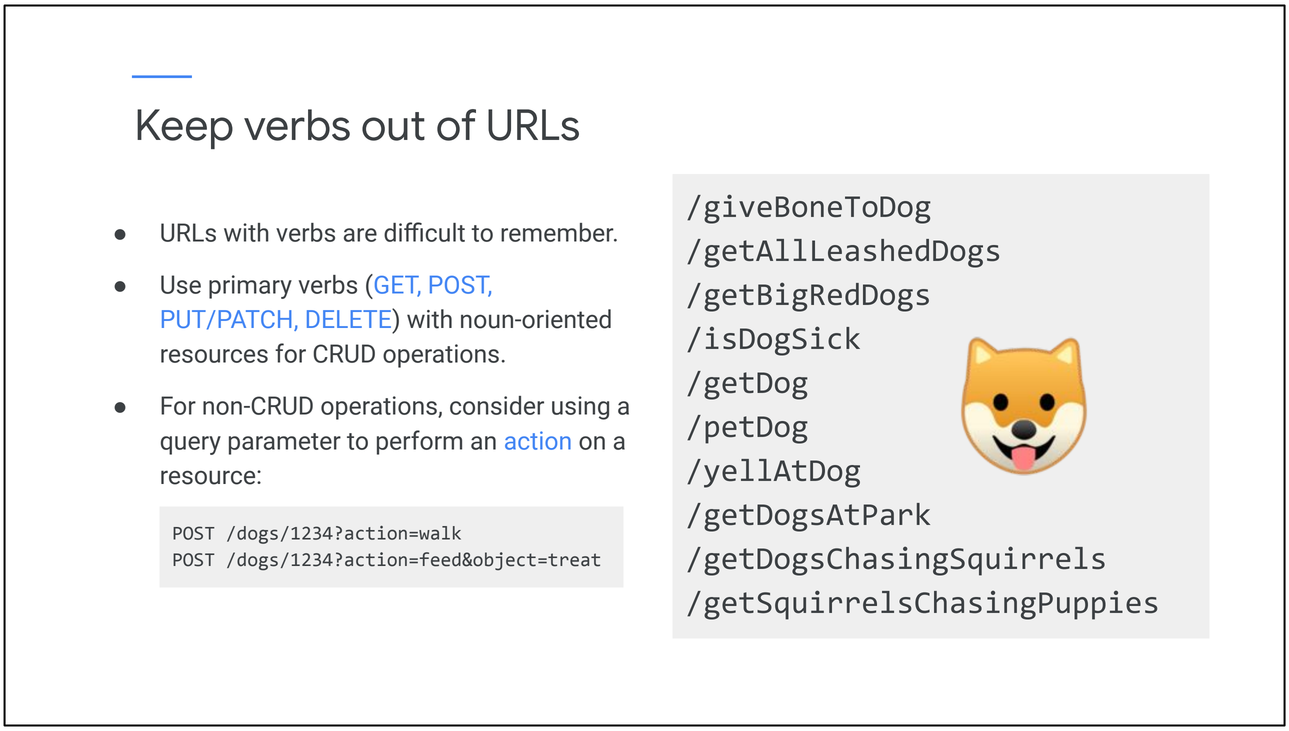
The **DELETE** method is used to delete a resource.

"DELETE /dogs/1234" would delete a specific dog.

"DELETE /dogs" would delete all dogs, or all dogs that match the search or filters provided. Most APIs don't allow a DELETE on a collection, because it would be too easy to accidentally delete everything in the collection.

DELETE updates the state by removing resources, so it isn't safe.

A second identical delete has no effect, since the resource was already deleted, so DELETE is idempotent.



Keep verbs out of URLs

- URLs with verbs are difficult to remember.
- Use primary verbs (GET, POST, PUT/PATCH, DELETE) with noun-oriented resources for CRUD operations.
- For non-CRUD operations, consider using a query parameter to perform an action on a resource:

```
POST /dogs/1234?action=walk
POST /dogs/1234?action=feed&object=treat
```

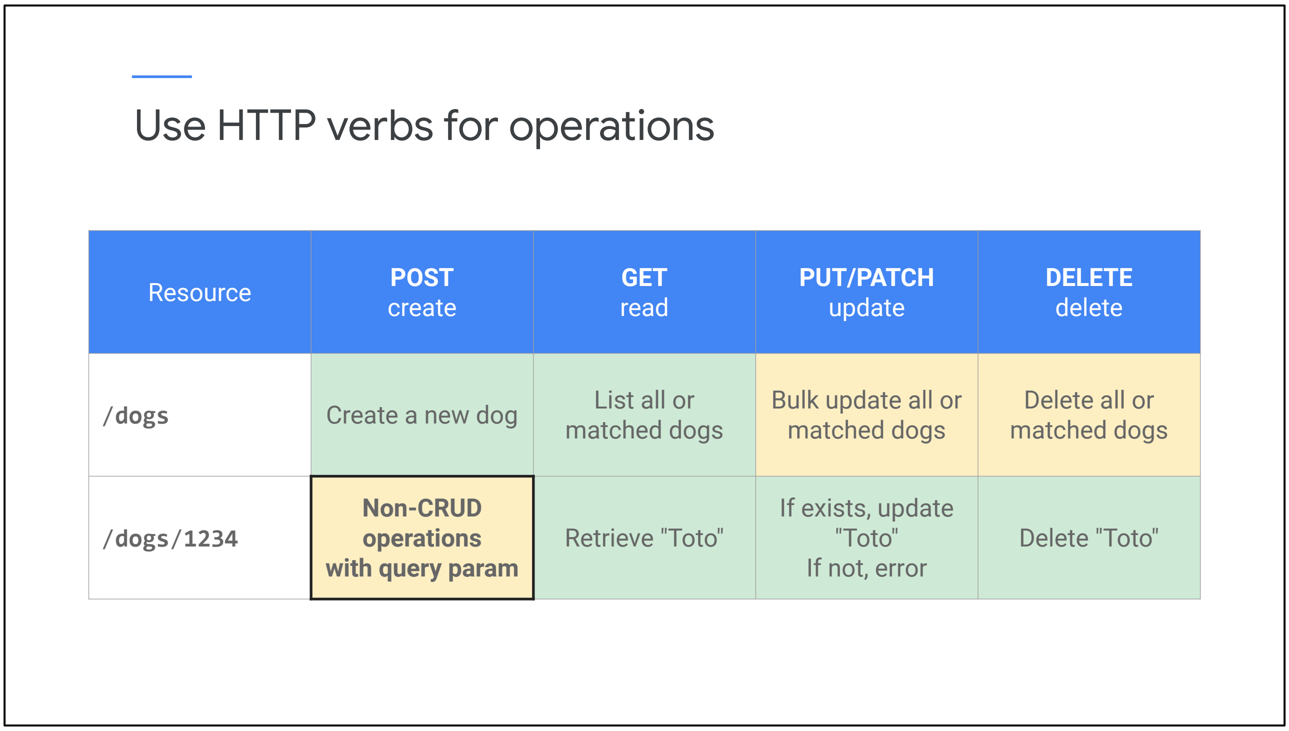
```
/giveBoneToDog
/getAllLeashedDogs
/getBigRedDogs
/isDogSick
/getDog
/petDog
/yellAtDog
/getDogsAtPark
/getDogsChasingSquirrels
/getSquirrelsChasingPuppies
```

If you have focused on SOAP services in the past, you may not be used to thinking first about resources, and then choosing operations that can be performed upon them, as we recommend for REST APIs. SOAP specifies operations in the URL, but this is an anti-pattern for REST APIs.

Operation names in URLs tend to have inconsistent patterns, making the calls difficult to remember and the API hard to use and learn.

For CRUD operations, we recommend that you use the primary HTTP verbs to operate on the resource.

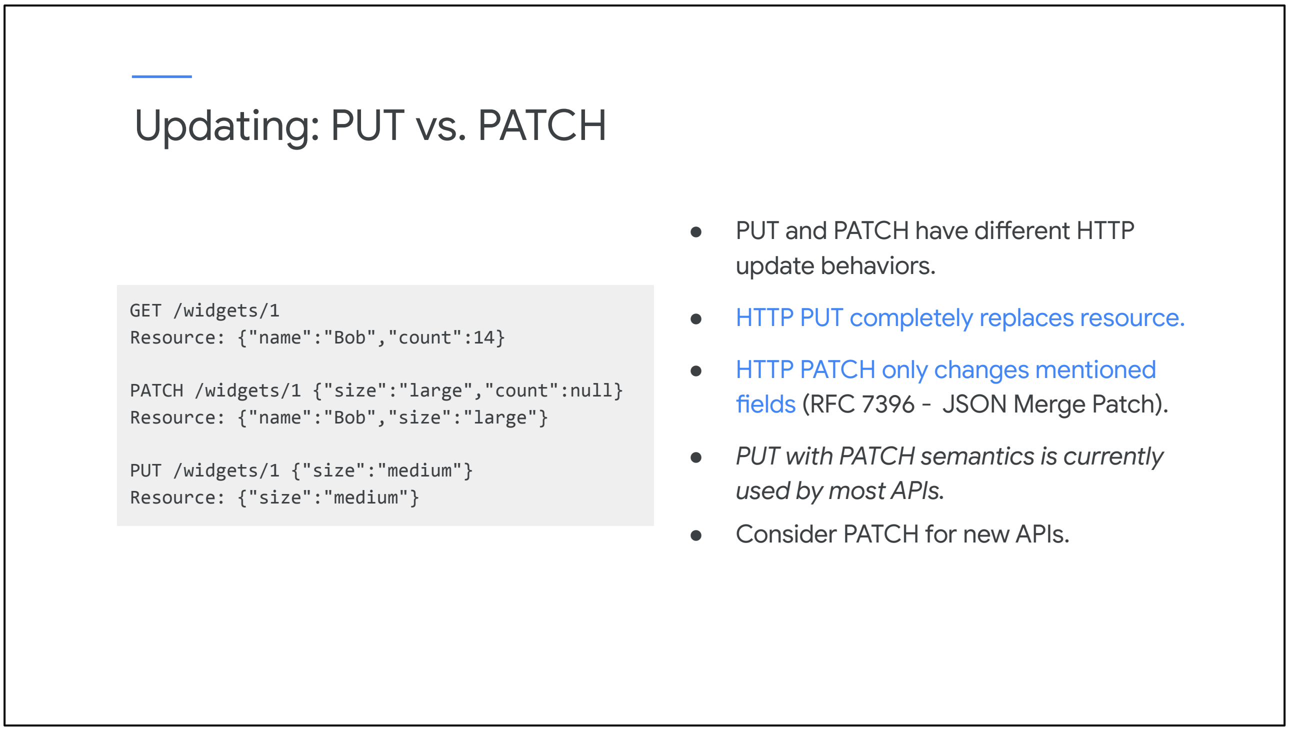
Operations will occasionally not fit into the CRUD model. For these cases, you can consider using POST with an action or operation query parameter to specify the operation on the resource. For example, "POST /dogs/1234" with an action parameter set to walk could represent walking the dog.



Use HTTP verbs for operations

Resource	**POST** create	**GET** read	**PUT/PATCH** update	**DELETE** delete
`/dogs`	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
`/dogs/1234`	**Non-CRUD operations with query param**	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

Here's the updated operation matrix: "POST /dogs/1234" with an action or operation query parameter can be used for non-CRUD operations.



Updating: PUT vs. PATCH

```
GET /widgets/1
Resource: {"name":"Bob","count":14}

PATCH /widgets/1 {"size":"large","count":null}
Resource: {"name":"Bob","size":"large"}

PUT /widgets/1 {"size":"medium"}
Resource: {"size":"medium"}
```

- PUT and PATCH have different HTTP update behaviors.
- HTTP PUT completely replaces resource.
- HTTP PATCH only changes mentioned fields (RFC 7396 - JSON Merge Patch).
- *PUT with PATCH semantics is currently used by most APIs.*
- Consider PATCH for new APIs.

The HTTP PUT and PATCH verbs are both used to update resources, but for HTTP they have different behaviors.

PUT completely replaces a resource with the provided payload.

PATCH should only be used to modify a subset of the resource. The JSON Merge Patch specification specifies how PATCH can be used for partial updates. Fields with a value are added or modified, and fields specified as null are deleted. Fields not mentioned in the PATCH request are not modified.

Let's look at an example.

Performing a GET on the widget with id of 1, we see that the widget's name is Bob, and its count is 14.

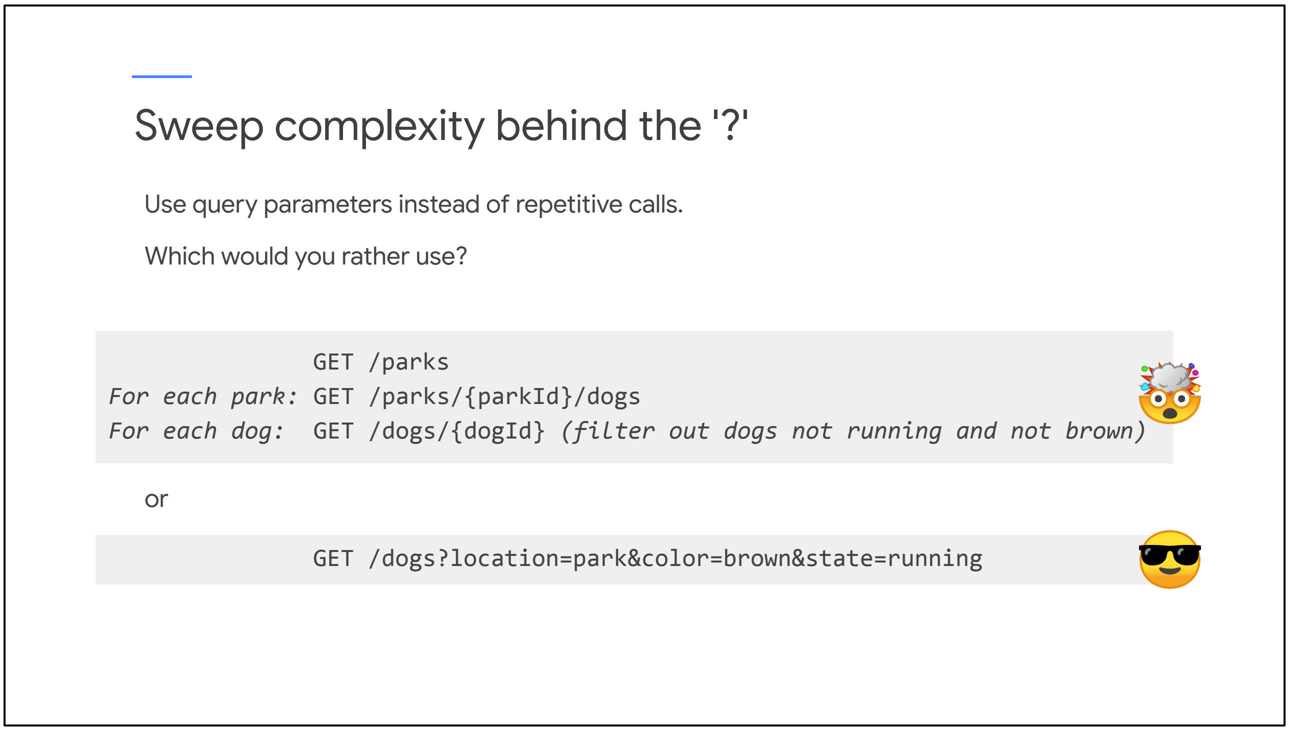
The PATCH call updates resource 1, setting the size field to large, and removing the count field since it is null. The name field is unchanged.

The PUT call replaces the entire resource with the request payload.

When you look at public REST APIs, most tend to use the PUT verb, even though they are actually using the semantics of HTTP PATCH, where fields that are not present in the request payload are not modified. Although this is not the standard HTTP use of PUT, it is very common in REST APIs.

When you are creating your own APIs, consider using PATCH for partial updates.

However, it helps your app developers if you maintain consistency for your APIs, so you may decide to use PUT for partial updates if other APIs are also using PUT in this way.



Sweep complexity behind the '?'

Use query parameters instead of repetitive calls.

Which would you rather use?

```
               GET /parks
For each park: GET /parks/{parkId}/dogs
For each dog:  GET /dogs/{dogId} (filter out dogs not running and not brown)
```
🤯

or

```
               GET /dogs?location=park&color=brown&state=running
```
😎

When we offer complex functionality in our APIs, we need to think about the app and developer experience.

For example, we need to find dogs based on factors like color, location, and action being performed. A simple design using only standard URL resources would allow this ability. But what would the app developer experience be?

To find brown dogs running in a park, the developer would first need to get all parks. Depending upon how many parks there are, it might take many API calls. Once she had all the parks, the developer would make a request for each park to find all the dogs in the park. She would then need to retrieve each dog found in each park, and keep a list of only those dogs that are running and are brown.

Obviously, this solution would require lots of programming for the developer. Even worse, it could result in a huge number of round trips as she retrieved all the dogs one by one. The user would have to wait a long time to get the results.

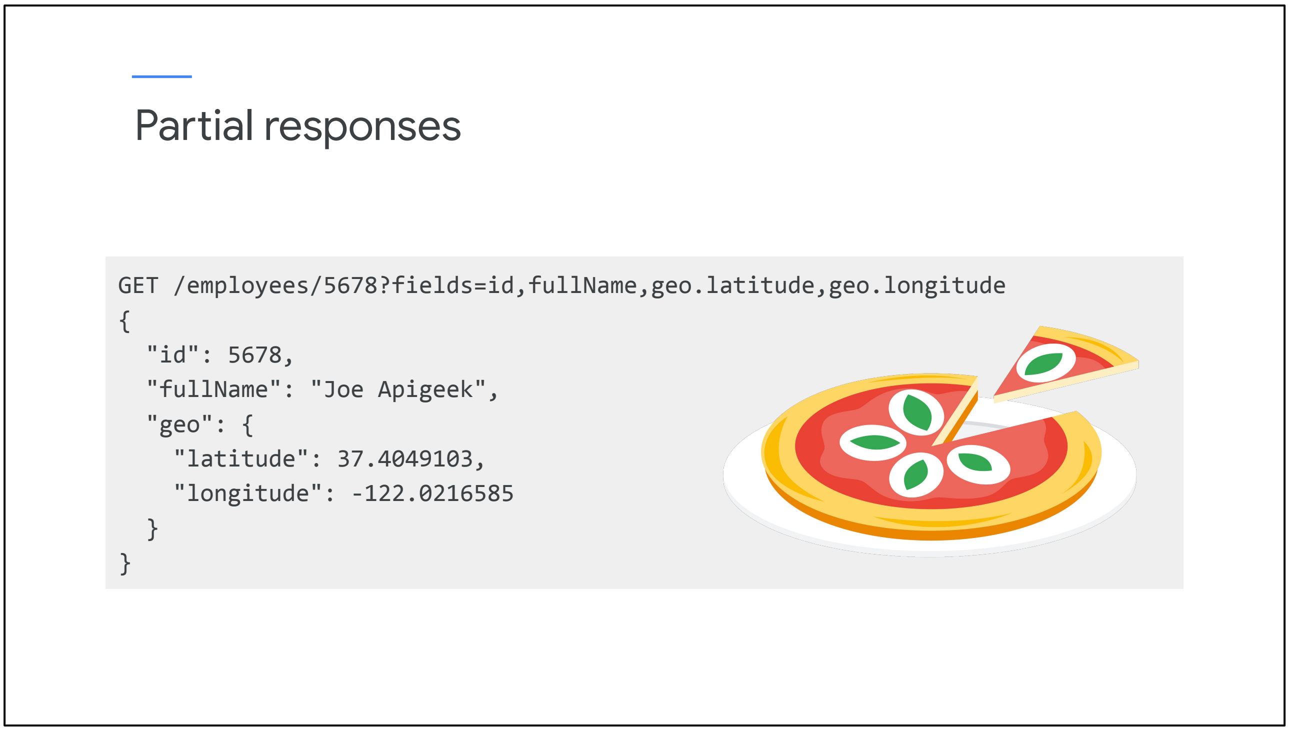
Now look at the bottom request, which can be viewed as "give me all the dogs that are in a park, and are brown, and are running."

Which pattern would you rather use? And which is more likely to perform well?

Coming up with the second solution requires the API developer to think like the app developer, and think about the experience of a user of an app that consumes your

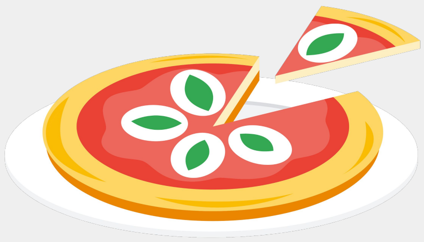
API. It is worth your time to think and design your APIs using this mindset.

Also, APIs may be used by apps in ways you didn't originally expect. You can add features to your APIs to address needs as you learn of them, either when requested by your app developers, or by analyzing API call patterns.



Partial responses

```
GET /employees/5678?fields=id,fullName,geo.latitude,geo.longitude
{
  "id": 5678,
  "fullName": "Joe Apigeek",
  "geo": {
    "latitude": 37.4049103,
    "longitude": -122.0216585
  }
}
```



Another way to deliver an optimized experience to your app developers is to allow the selection of partial responses. Instead of returning the entire response every time, you can give the developer control over which fields to return.

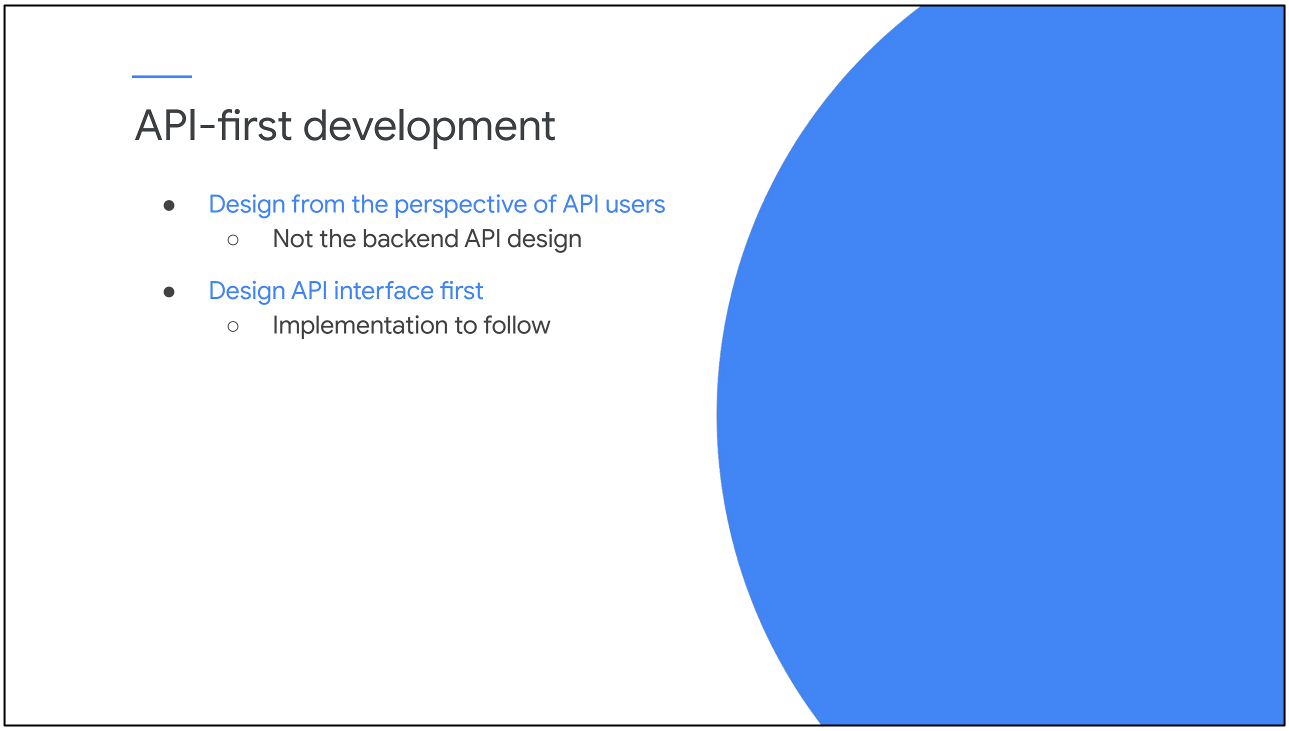
This feature should definitely be considered for APIs that are designed for usage in low bandwidth apps, like mobile apps, or for APIs with large response payloads.

In this example, a list of comma-separated fields is specified using a query parameter, and only the requested fields are returned in the response.

API-First Development

This lecture introduces the concept of API-first development.

First, the obvious question: what is API-first development?



API-first development

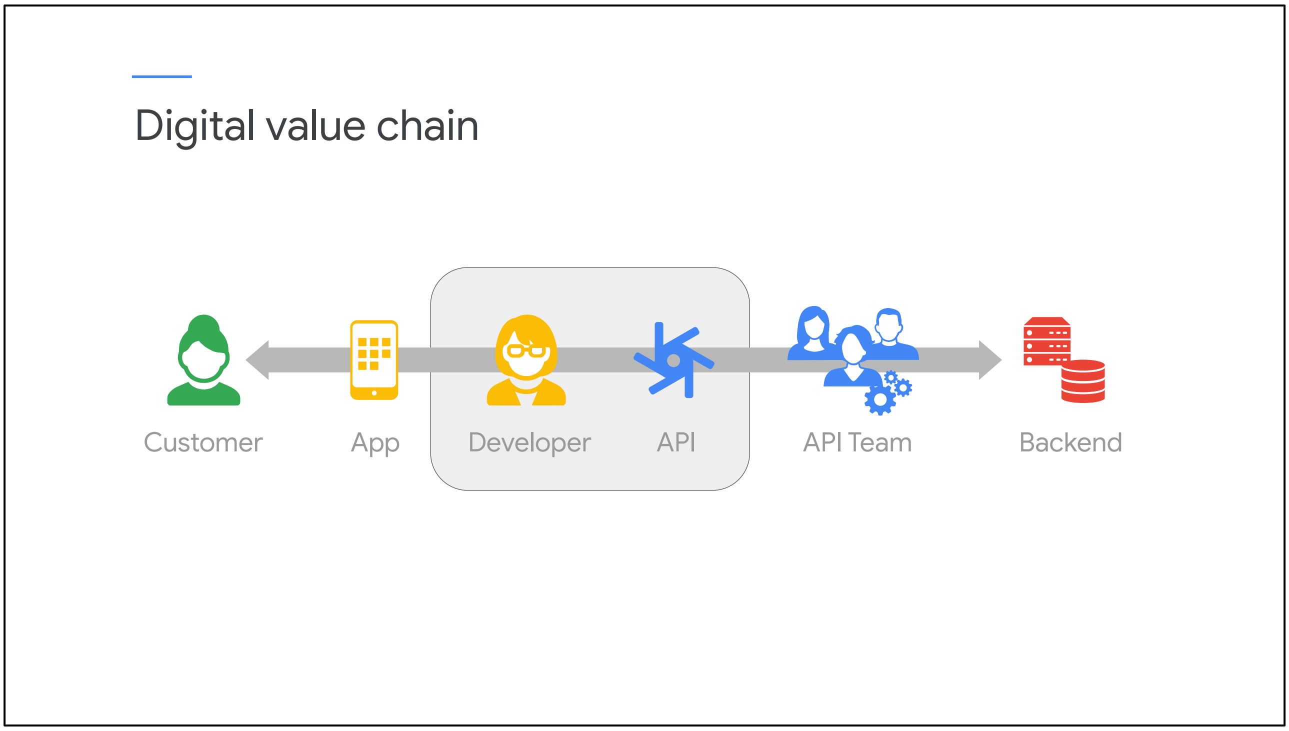
- Design from the perspective of API users
  - Not the backend API design
- Design API interface first
  - Implementation to follow

API-first development is a strategy where your APIs are designed by first focusing on the needs of the app developers who will consume your API.

Your API will be more successful if your app developers enjoy using your API and it solves their problems.

By focusing on the needs of your app developers, you may find that your API design should differ significantly from the design of your backend APIs. It may be useful to think of your backend APIs as building blocks for the API you want to create.

Your API interface should also be designed before you start implementing your APIs. This helps you avoid making questionable design decisions based on what happens during the implementation process.



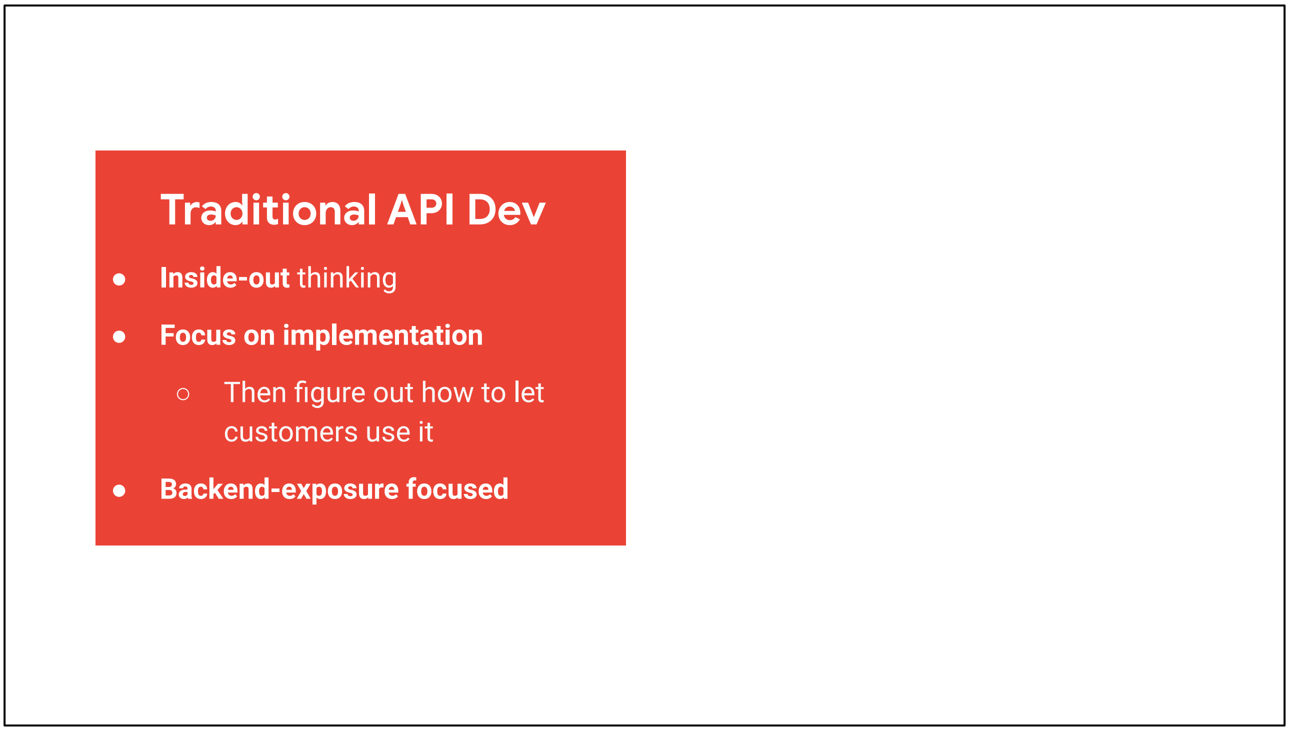
Digital value chain

Let's return to the digital value chain.

App developers build apps that deliver connected experiences for end users. These experiences drive value for the company.

APIs built by the API team will be used to power those applications.

API-first development recognizes that your app developers will be most effective if those APIs are crafted specifically for their needs.

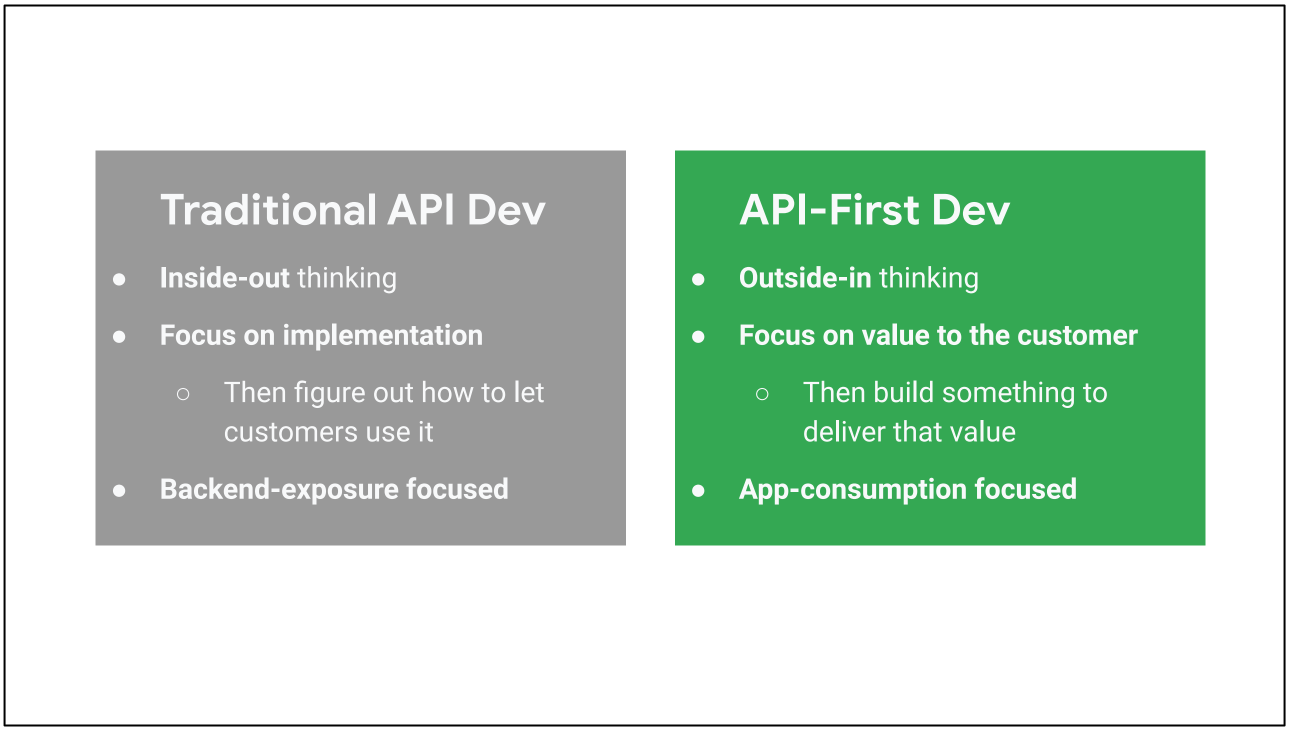


Traditional API Dev

- **Inside-out** thinking
- **Focus on implementation**
  - Then figure out how to let customers use it
- **Backend-exposure focused**

Traditional API development has often been driven by inside-out thinking: build services first, and then worry about how to expose them to the customer.

A traditional goal may sound something like: "There are 20 APIs we use in our company, and we want to expose them to the world." This is focusing on the implementation of the API first, instead of focusing on the needs of the application developers who will be the customers of your APIs. Even though you may be able to check a box saying you made 20 APIs available, you run the risk of having APIs that no one wants to use or that perform poorly when used in applications.



Traditional API Dev

- **Inside-out** thinking
- **Focus on implementation**
  - Then figure out how to let customers use it
- **Backend-exposure focused**

API-First Dev

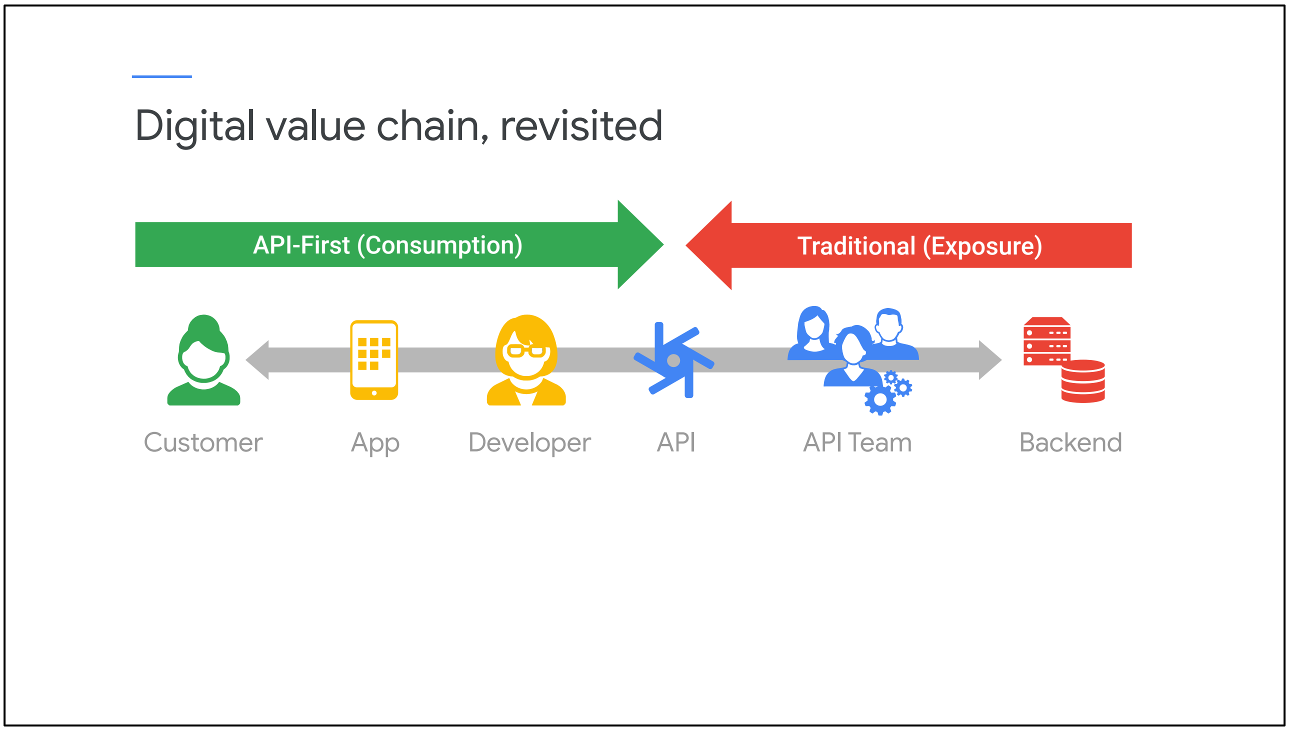
- **Outside-in** thinking
- **Focus on value to the customer**
  - Then build something to deliver that value
- **App-consumption focused**

In contrast, if you adopt an API-first approach, you are basing your API design decisions on the needs of your potential customers.

This is outside-in thinking: designing with value for the customer as your primary focus, and then building something to deliver that value.

If you understand what your app developers want and can design an API that solves their problems and is enjoyable to use, your API has a much better chance of being successful.

Concentrate your API design efforts on how your API will be consumed by your app developers and their applications.

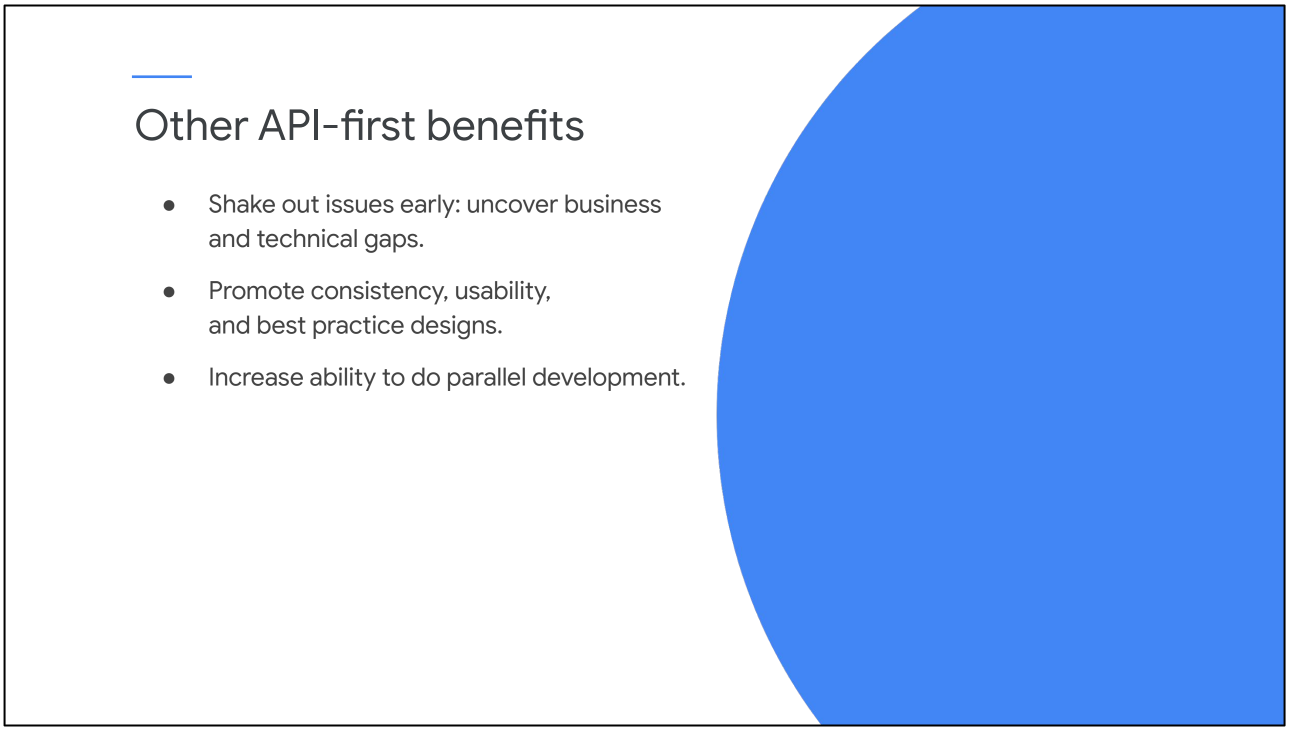


Digital value chain, revisited

Back to the digital value chain.

If your primary focus is on simply exposing what you already have, or you are designing your APIs based on your existing backend implementation, you aren't designing with the customer in mind.

When your design is primarily informed by the needs and desires of your app developers and their apps and customers, focusing on usability and customer use cases, your APIs have a better chance to succeed.



Other API-first benefits

- Shake out issues early: uncover business and technical gaps.
- Promote consistency, usability, and best practice designs.
- Increase ability to do parallel development.

There are other benefits that come with adopting an API-first approach.

By designing your API contracts before you implement the APIs, you can have customers and stakeholders uncover business and technical gaps in your APIs before you develop them. You won't spend time building features that are not needed, and you won't design backend systems without understanding how they will be used.

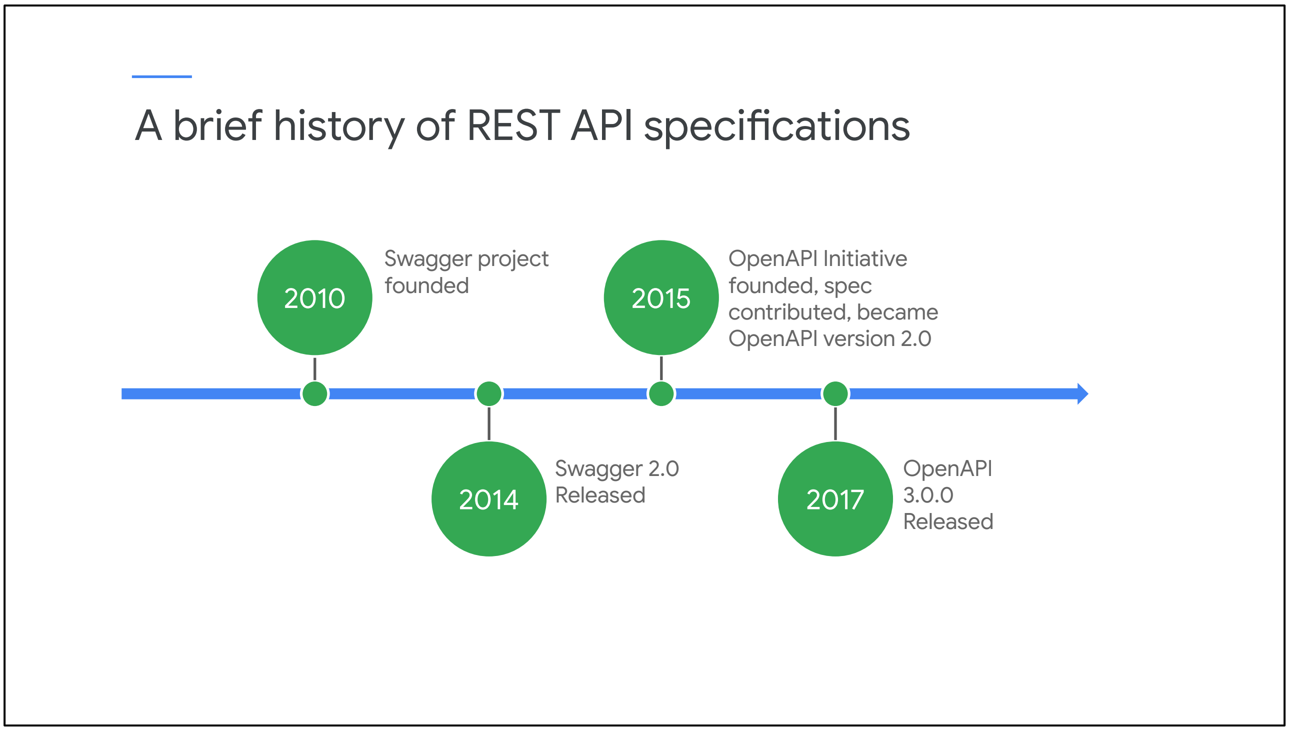
This early review cycle also allows your APIs to be reviewed for consistency, usability, and best practices. Consistency across your APIs tends to increase adoption and give app developers confidence in your API program.

Designing the API contract early also increases the ability to do parallel development. Apps using the APIs can be designed, and use cases can be validated, while your API implementation is in progress.

OpenAPI Specs

This lecture will introduce OpenAPI Specifications, the preferred way to document your RESTful APIs.

We will learn about the benefits of OpenAPI specifications, as well as how OpenAPI specs can be used with Apigee.



A brief history of REST API specifications

In the early days, JSON-based APIs were documented using WADL, which stands for Web Application Description Language. WADL served a purpose similar to WSDL, or Web Services Description Language, which is used to describe SOAP-based services. Like WSDL, WADL is relatively difficult to use: it is XML-based, complex, and not very human-readable.

Swagger, introduced in 2010, was a big step in the right direction, specifying JSON-formatted files to describe the interface for RESTful APIs. This format was easier to understand and build.

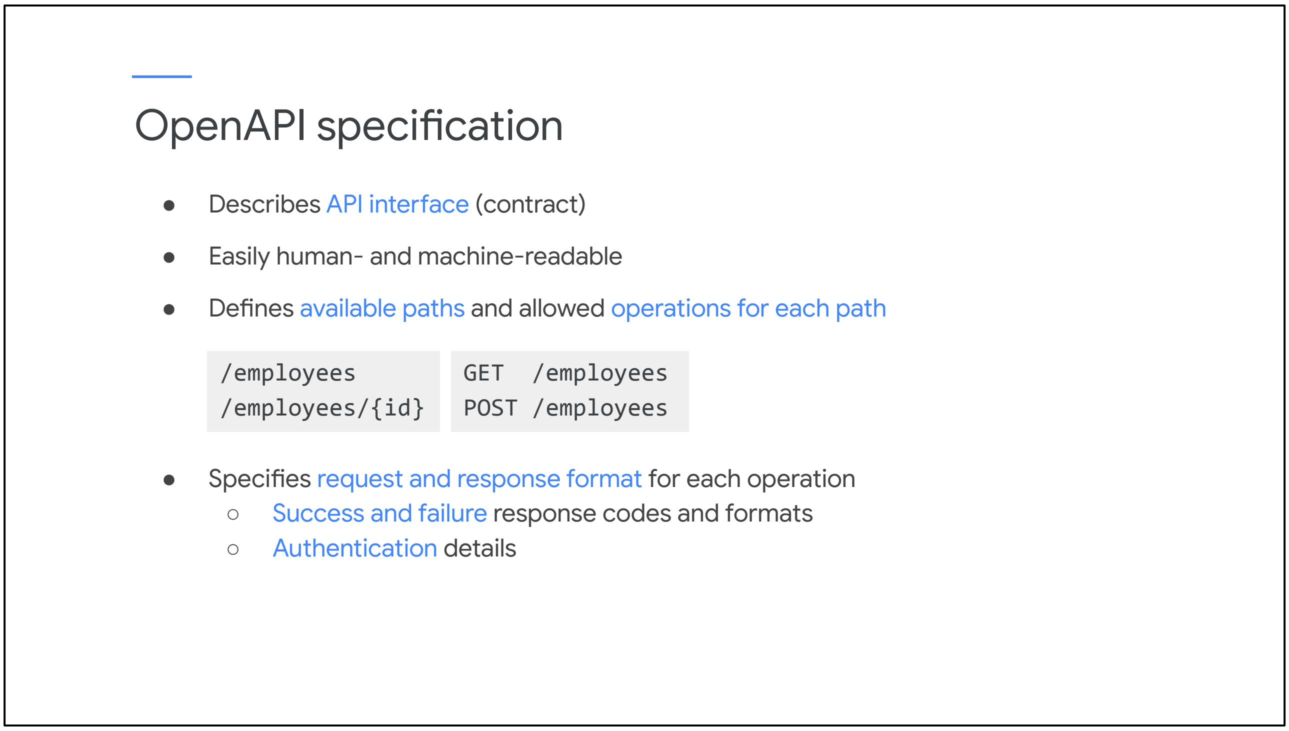
Swagger 2.0 introduced support for the YAML format. YAML is a recursive acronym which stands for YAML Ain't Markup Language. YAML is more human-readable than JSON.

Swagger 2.0 also provided the ability to describe all aspects of a RESTful API within a single file.

Swagger is a proprietary format, but the format was donated to the OpenAPI Initiative so that the format could be fully open-sourced. This became OpenAPI version 2.

OpenAPI 3 further improved the structure and reusability of the components in an OpenAPI specification, and also added features like examples and callbacks.

Apigee currently supports OpenAPI version 2 and 3.



OpenAPI specification

- Describes API interface (contract)
- Easily human- and machine-readable
- Defines available paths and allowed operations for each path

```
/employees          GET  /employees
/employees/{id}     POST /employees
```

- Specifies request and response format for each operation
  - Success and failure response codes and formats
  - Authentication details

OpenAPI specifications are a great way to document your REST APIs.

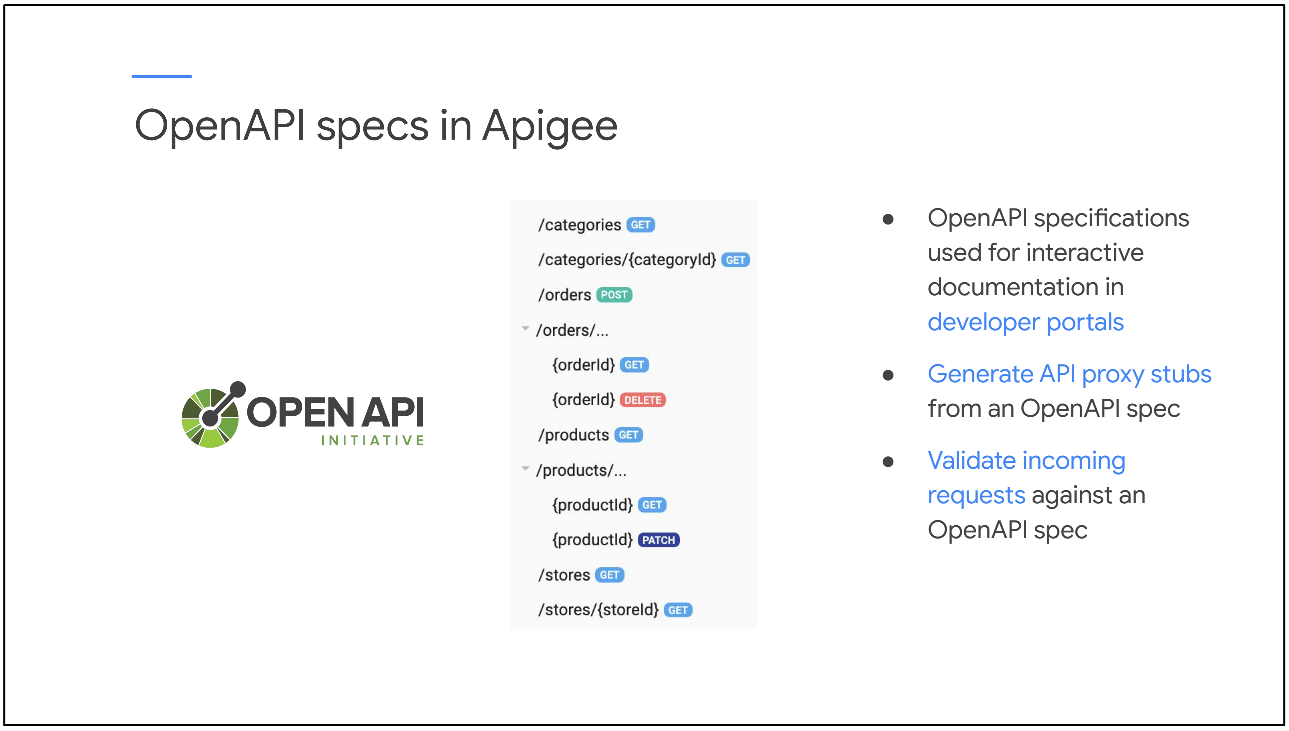
OpenAPI specs describe the API interface, or contract, that defines how apps can use the API.

An OpenAPI spec is laid out in a format that promotes readability and editability.

OpenAPI specs define the available paths to resources and the allowed operations for each path.

The request and response formats for all API calls should be documented in the spec, including success and failure response codes and payload formats for all failure types.

Authentication details for the API should also be described in the spec.



OpenAPI specs in Apigee

OPEN API INITIATIVE

```
/categories GET
/categories/{categoryId} GET
/orders POST
/orders/...
   {orderId} GET
   {orderId} DELETE
/products GET
/products/...
   {productId} GET
   {productId} PATCH
/stores GET
/stores/{storeId} GET
```

- OpenAPI specifications used for interactive documentation in developer portals
- Generate API proxy stubs from an OpenAPI spec
- Validate incoming requests against an OpenAPI spec

OpenAPI specifications are used to provide interactive documentation for application developers in the developer portal. Being able to try your API streamlines the learning process for your app developers.

We'll learn more about the developer portal in a future lecture.

OpenAPI specifications can also be used to generate Apigee API proxy stubs. The generated proxy stubs will include flows for all resources and operations documented in the API specification.

You can also use an OpenAPI spec validation policy in your API proxy to validate incoming requests against an OpenAPI specification. Any requests that do not match the spec could be rejected.

We will learn more about API proxies in the next lectures.

Review: API-First and OpenAPI Specifications

Hansel Miranda
Course Developer, Google Cloud

You have learned about RESTful APIs, and what we should consider when designing them.

We discussed API-first development, which tells us to design our APIs from the outside in, focusing on app developers and their needs.

And you learned about OpenAPI specifications, and how they are used to document APIs.