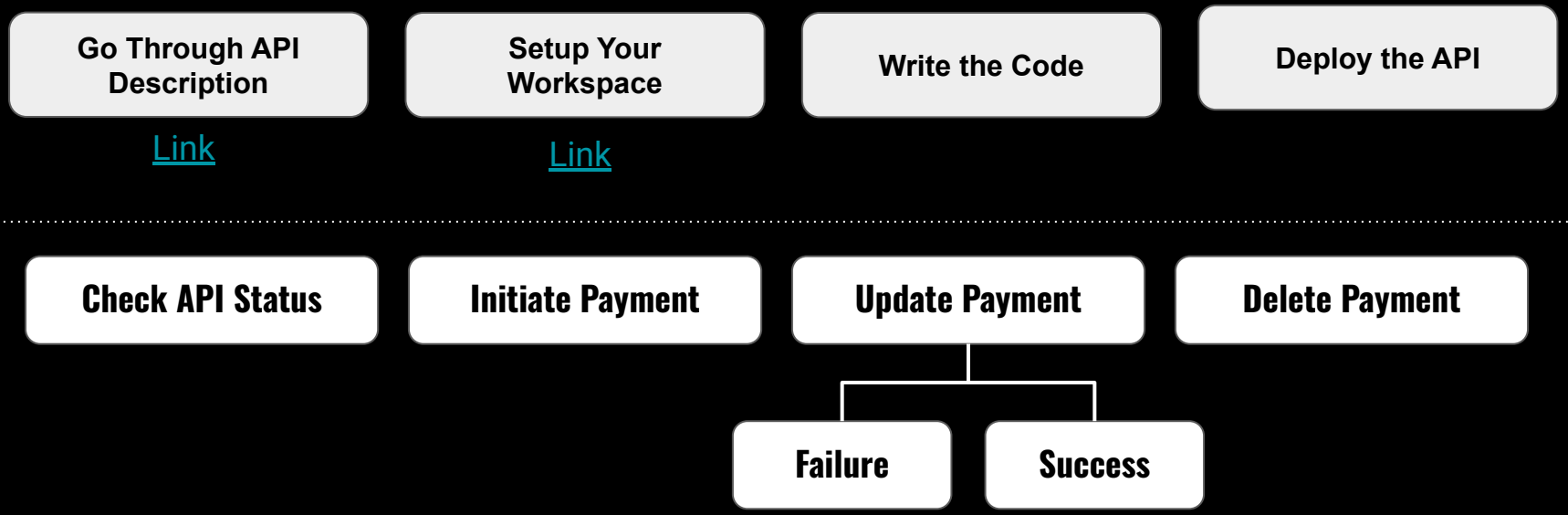
Designing APIs From Scratch

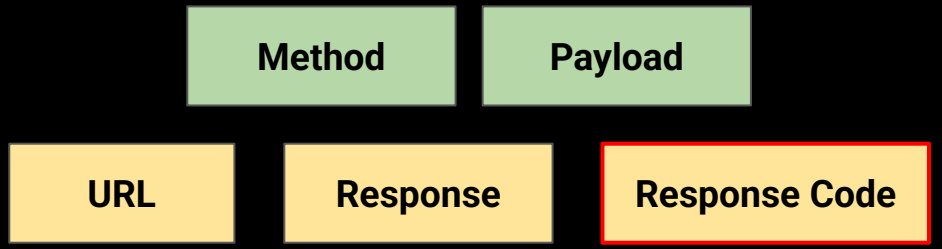
Payment Based API



Let’s continue with next steps

Validations

Story So Far



Using HTTP status codes to create expressive HTTP responses

What is HTTP status code?

How we classify them into groups?

How to use them to model our API responses?

What are Status Codes?

Status codes are those three-digit codes that we find in an HTTP **response**

They are three-digit numbers from 100 to 599 inclusive that indicate the high-level semantics of the response

Broadly speaking, we can think of these statuses as the success or failure of the request

Perhaps the best-known status code, and an example of a failure, is 404 Not Found, which of course means that the resource you're after isn't there, or you don't have authorization to know if it exists

What are HTTP status Codes?

HTTP response status codes are used to indicate the outcome of processing an HTTP request

For example, the 200 status code indicates that the request was successfully processed

While the 500 status code indicates that an internal server error was raised while processing the request

Full list of response code can be found here: Link

Let’s look at the most commonly used codes and see how we apply them in our API designs

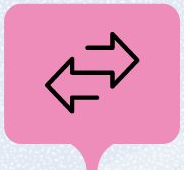
What are HTTP Status Codes?

1XX

Signals that an operation is in progress

2XX

Signals that a request was successfully processed



3XX

Signals that a resource has been moved to a new location

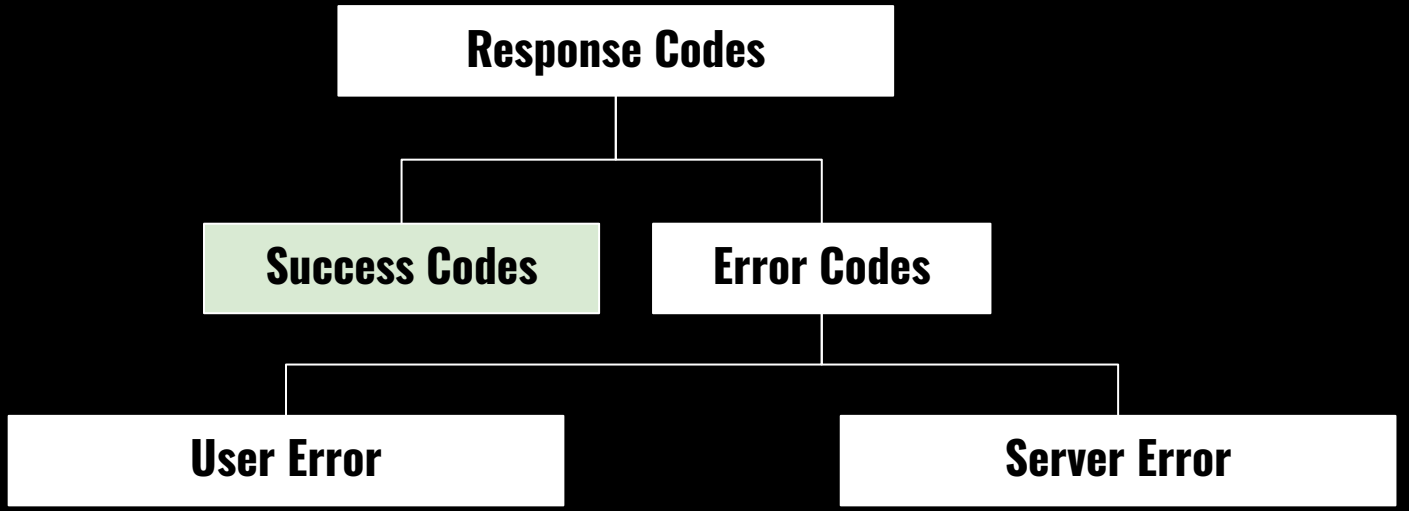
4XX

Signals that something was wrong with the request

5XX

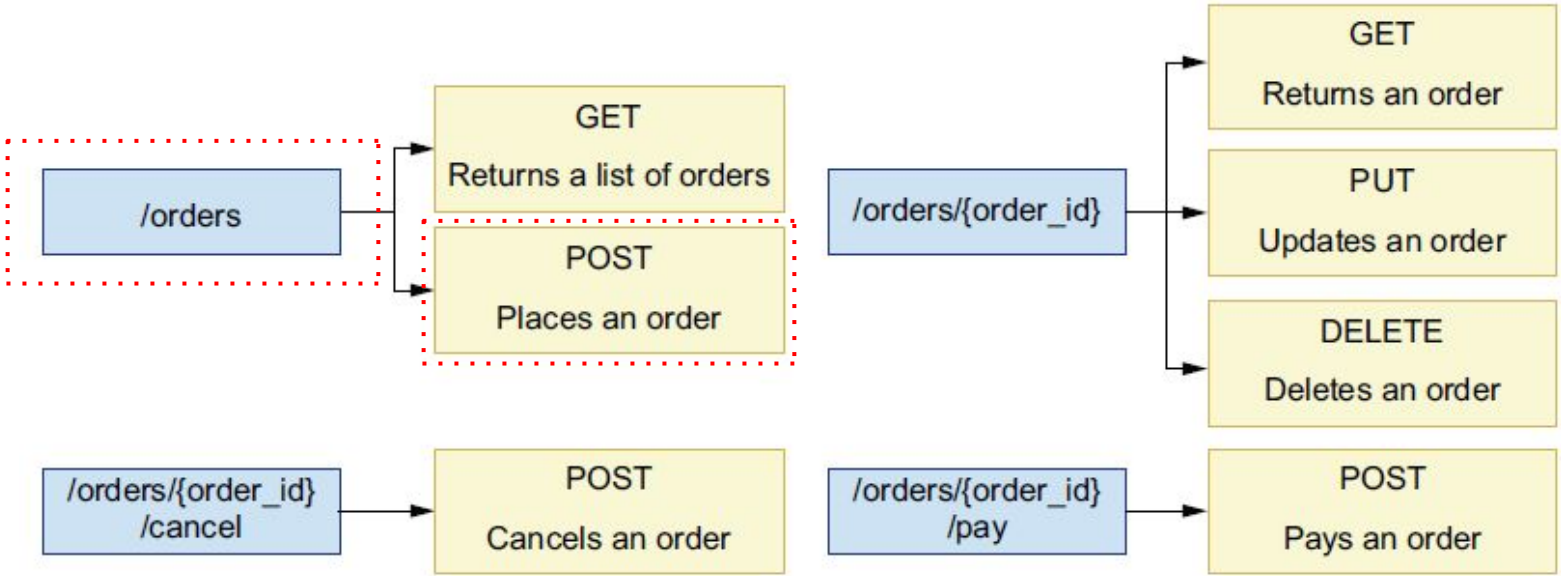
Signals that there was an error while processing the request

Using HTTP status codes to create expressive HTTP responses



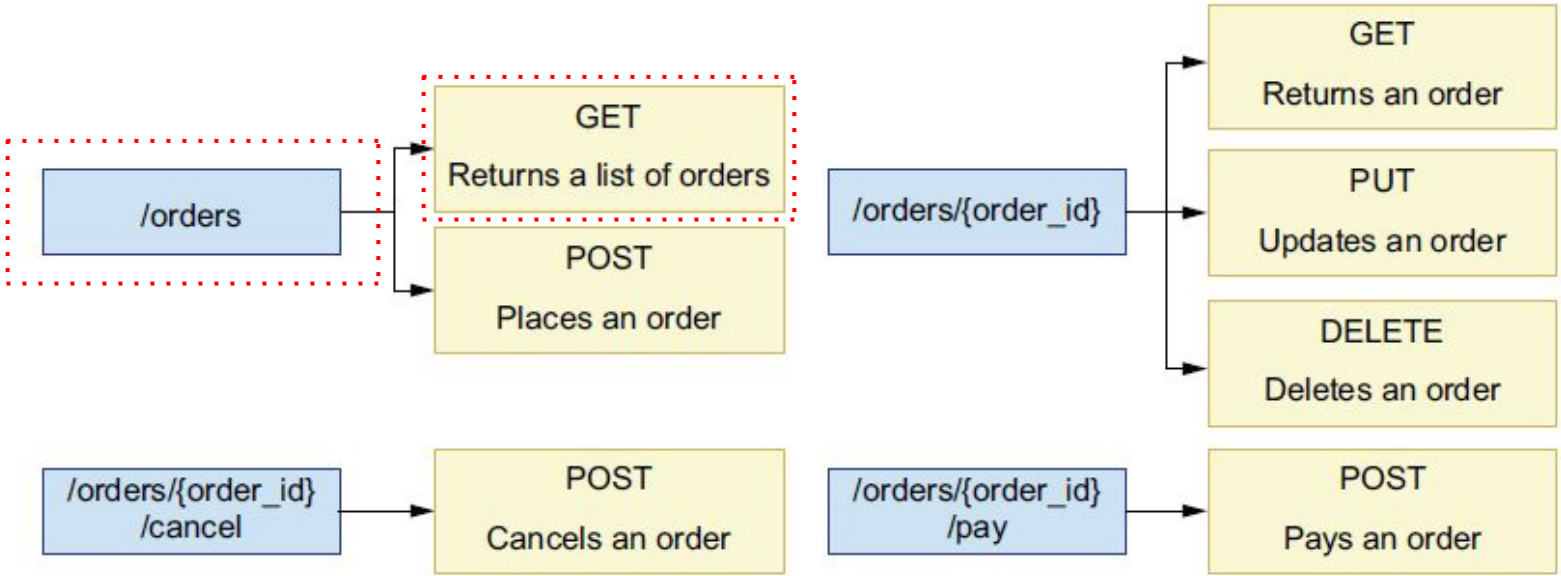
Status Code of Order API Endpoint: Create An Order

POST /orders: **201 (Created)**—Signals that a resource has been created



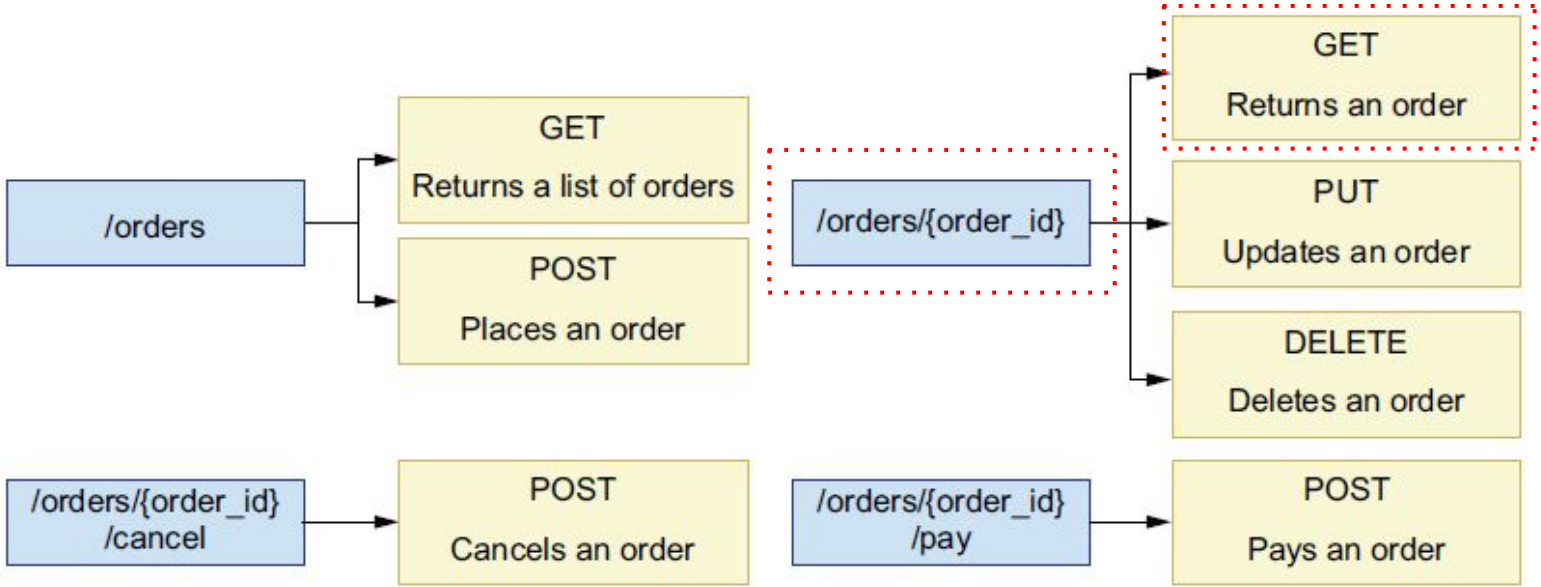
Status Code of Order API Endpoint: Fetch All Orders

GET /orders: **200 (OK)**—Signals that the request was successfully processed



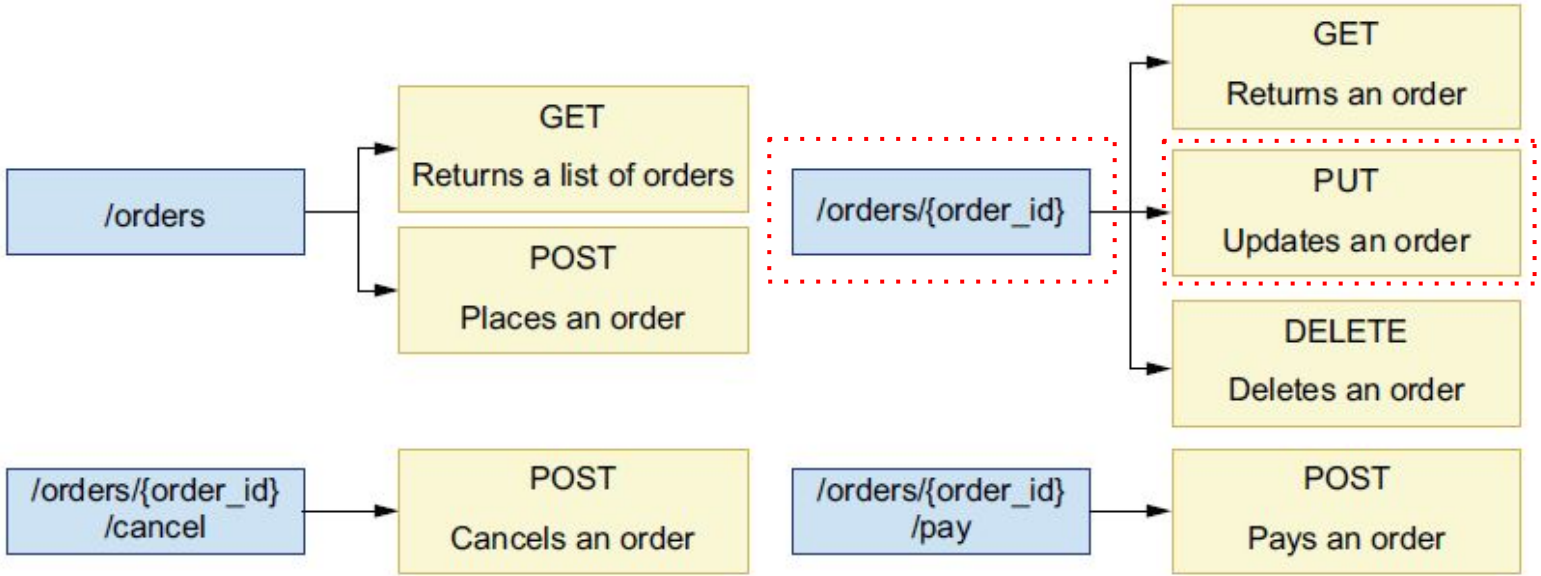
Status Code of Order API Endpoint: Fetch one Order

GET /orders/{order_id}: **200 (OK)**—Signals that the request was successfully processed



Status Code of Order API Endpoint: Update an Order

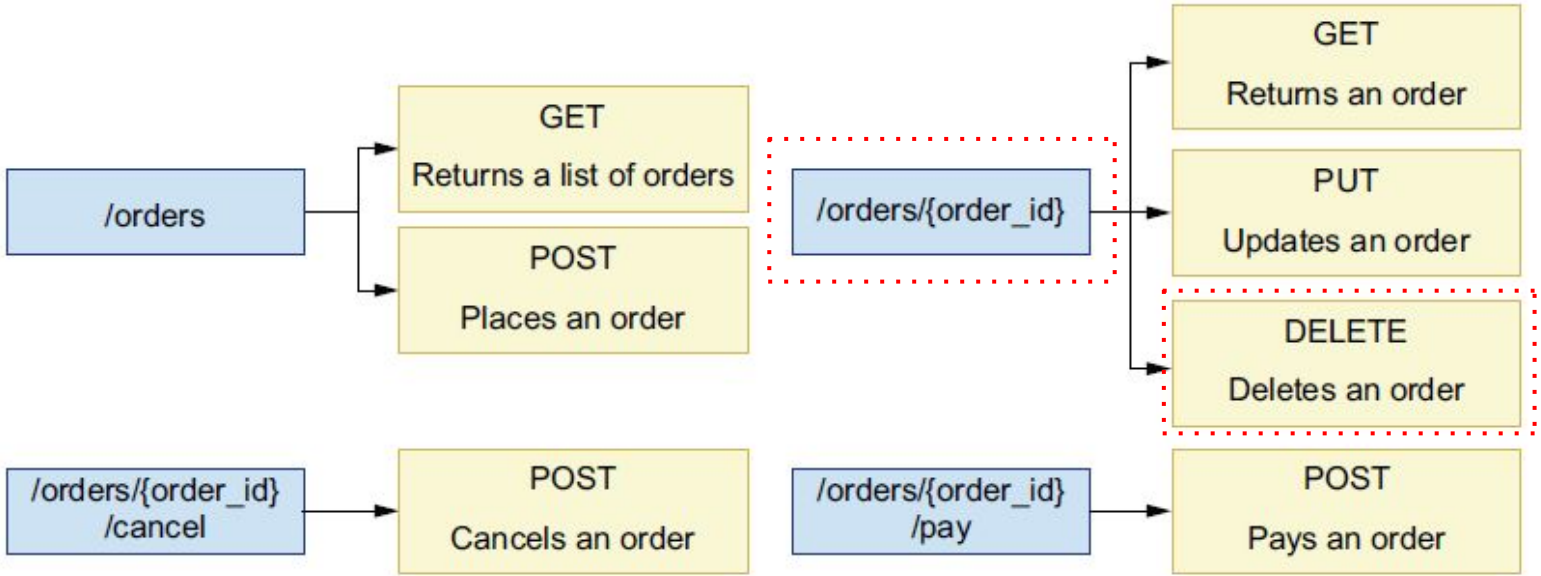
PUT /orders/{order_id}: **200 (OK)**—Signals that the resource was successfully updated



Status Code of Order API Endpoint: Delete An Order

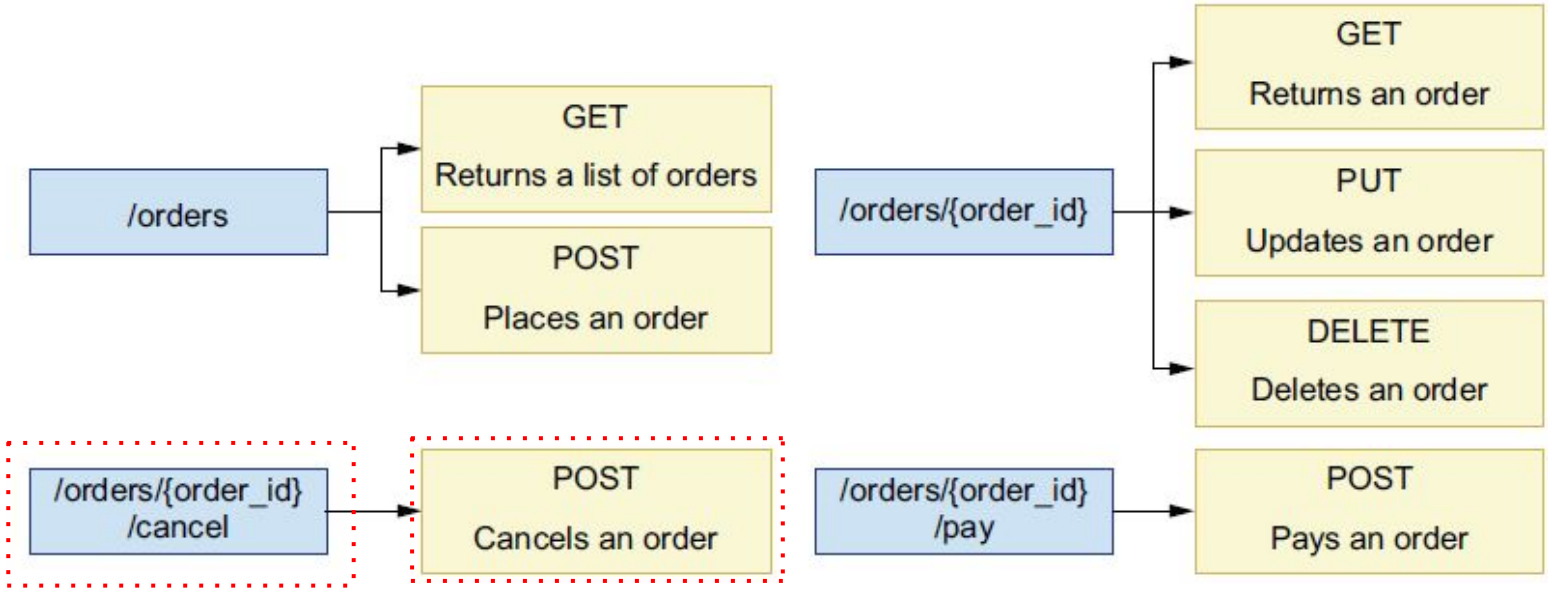
DELETE /orders/{order_id}: **204 (No Content)**—Signals that the request was successfully processed but no content is delivered in the response

Contrary to all other methods, a DELETE request doesn't require a response with payload



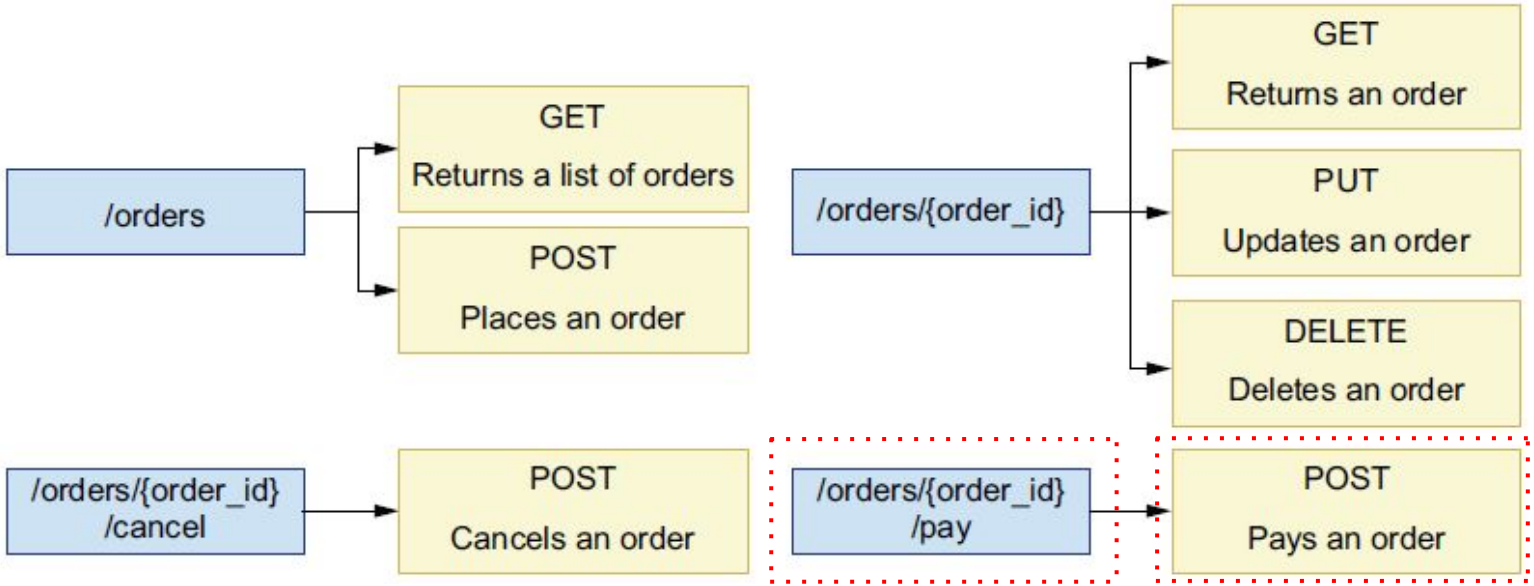
Status Code of Order API Endpoint: Cancel An Order

POST /orders/{order_id}/cancel: 200 (OK)—Although this is a POST endpoint, we use the 200 (OK) status code since we're not really creating a resource, and all the client wants to know is that the cancellation was successfully processed

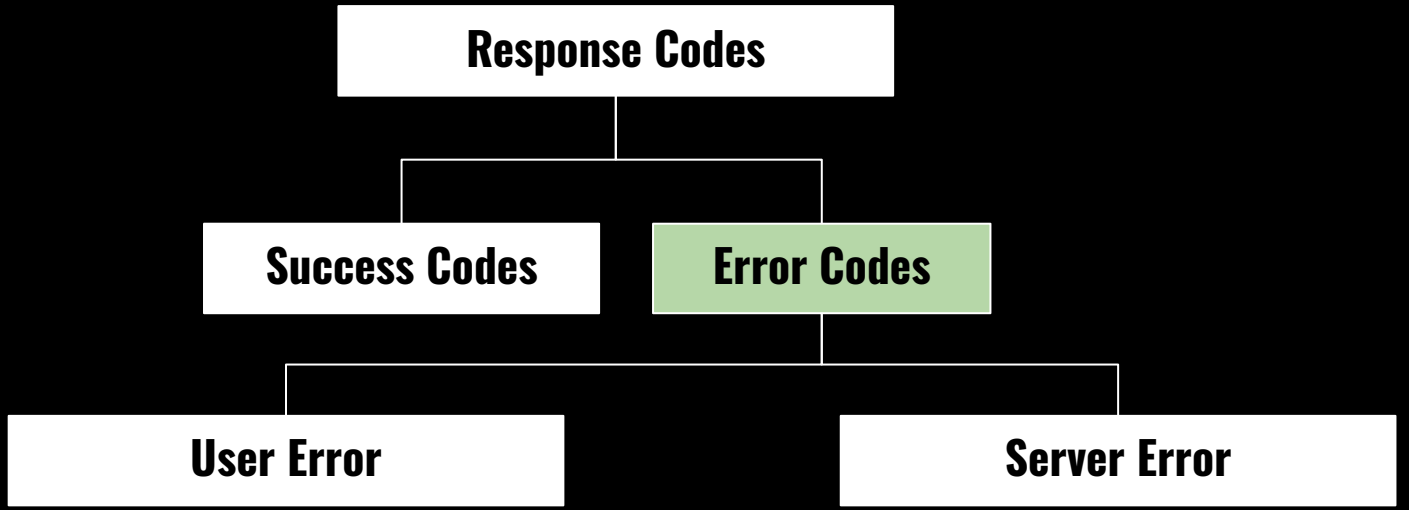


Status Code of Order API Endpoint: Pay For An Order

POST /orders/{order_id}/pay: **200 (OK)**—Although this is a POST endpoint, we use the 200 (OK) status code since we're not really creating a resource, and all the client wants to know is that the payment was successfully processed



Using HTTP status codes to create expressive HTTP responses



Status Code of Order API Endpoint: Error Codes

That's all good for successful responses, but what about error responses?

What kinds of errors can we encounter in the server while processing requests, and what kinds of HTTP status codes are appropriate for them?

User Error

Errors made by the user when sending the request, for example, due to a malformed payload, or due to the request being sent to a nonexistent endpoint

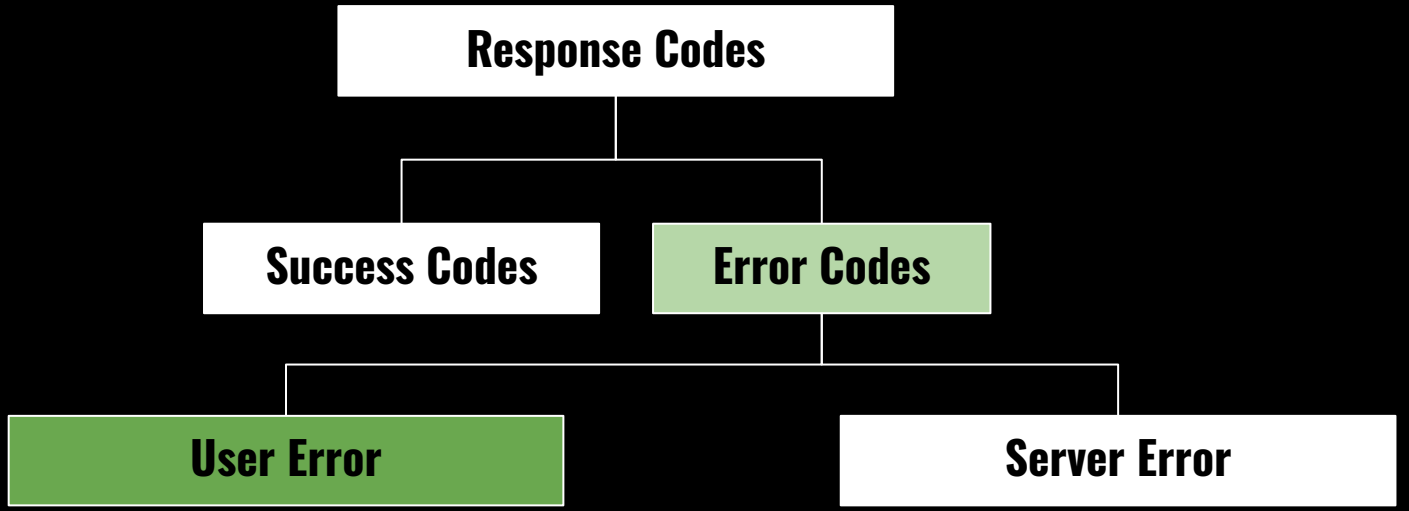
We address this type of error with an HTTP status code in the 4xx group

Server Error

Errors unexpectedly raised in the server while processing the request, typically due to a bug in our code

We address this type of error with an HTTP status code in the 5xx group

Using HTTP status codes to create expressive HTTP responses



User Error: Using HTTP status codes to report client errors in the request

An API client can make different types of errors when sending a request to an API

The most common type of error in this category is sending a malformed payload to the server

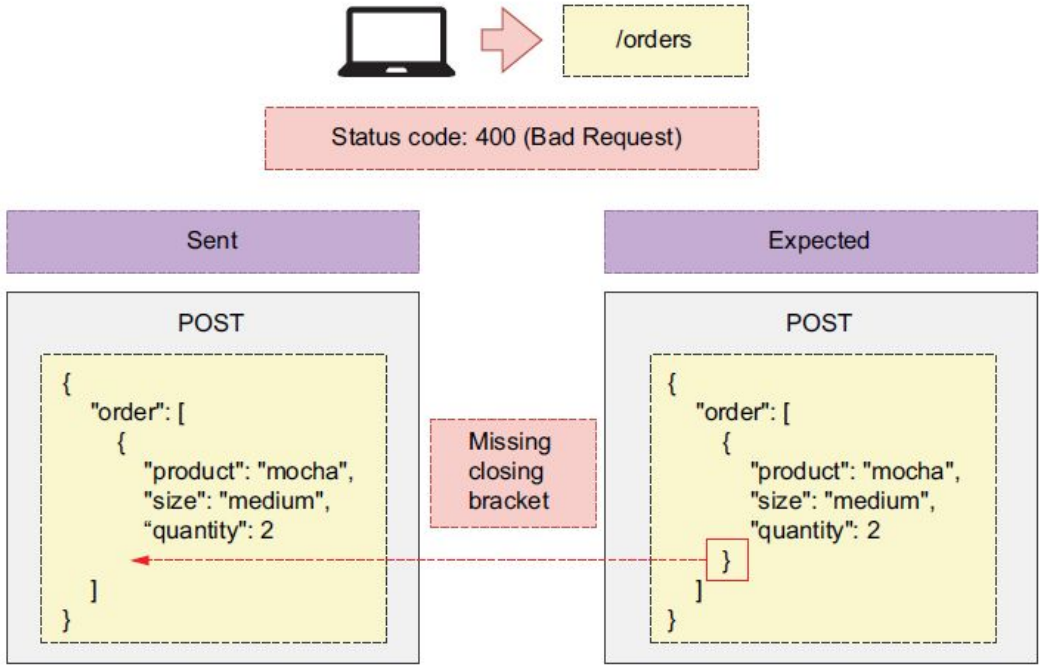
We distinguish two types of malformed payloads: **payloads with invalid syntax** and **unprocessable entities**

Using HTTP status codes to report client errors in the request: User Error-Invalid Request (400) - Syntactic Error

Payloads with invalid syntax are payloads that the server can neither parse nor understand

A typical example of a payload with invalid syntax is malformed JSON

As you can see in figure, we address this type of error with a **400 (Bad Request)** status code



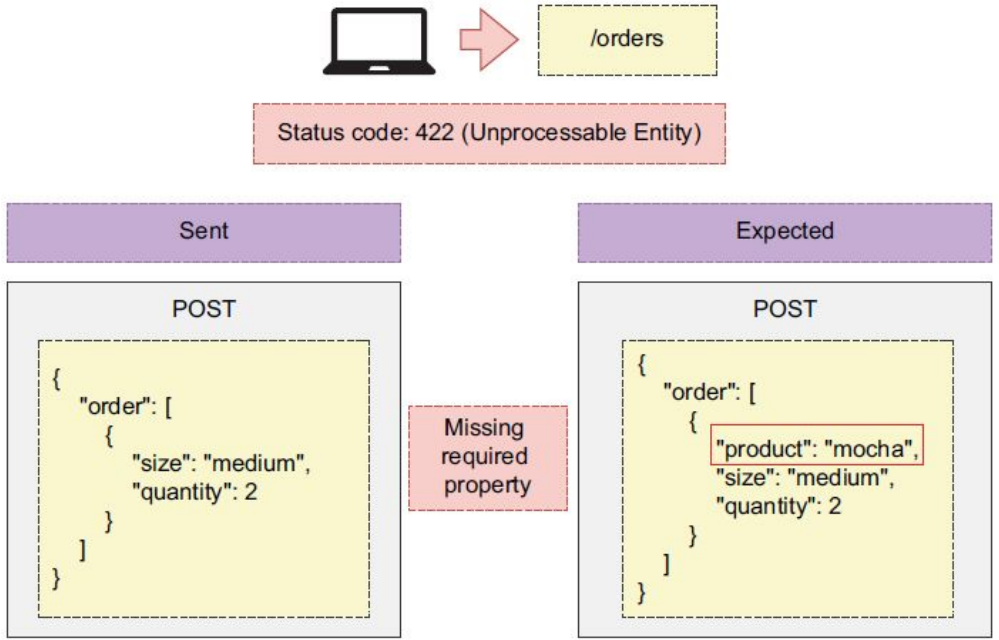
Using HTTP status codes to report client errors in the request: User Error-Valid Request, But Missing Required Field (422) - Semantic Error

Unprocessable entities are syntactically valid payloads that **miss a required parameter**, contain **invalid parameters**, or assign the **wrong value or type** to a parameter

To place an order: required params are product name, size and quantity

As you can see in figure, product field is missing & therefore is un-processable

We address this type of error with the **422 (Unprocessable Entity)** status code, which signals that something was wrong with the request and it couldn't be processed



Using HTTP status codes to report client errors in the request: User Error-Resource Does Not Exist (404)

Another common error happens when an API client requests a resource that doesn't exist

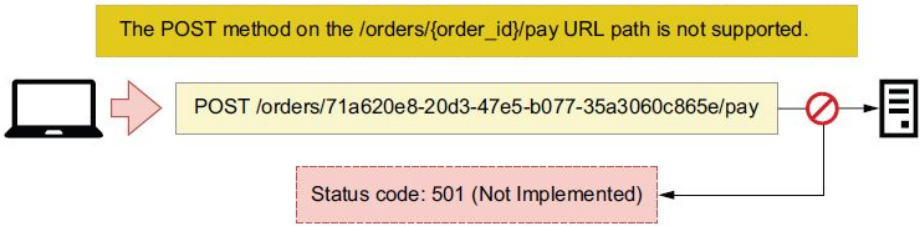
For example, we know that the GET /orders/{order_id} endpoint serves the details of an order. If a client uses that endpoint with a nonexistent order ID, we should respond with an HTTP status code signaling that the order doesn't exist. As you can see in figure, we address this error with the **404 (Not Found)** status code, which signals that the requested resource is not available or couldn't be found

Using HTTP status codes to report client errors in the request: User Error-Wrong Method (501, 405)

Another common error happens when API clients send a request using an HTTP method that is not supported

For example, if a user sent a PUT request on the /orders endpoint, we must tell them that the PUT method is not supported on that URL path

We can return a **501 (Not Implemented)** if the method hasn't been implemented but will be available in the future (i.e., we have a plan to implement it)



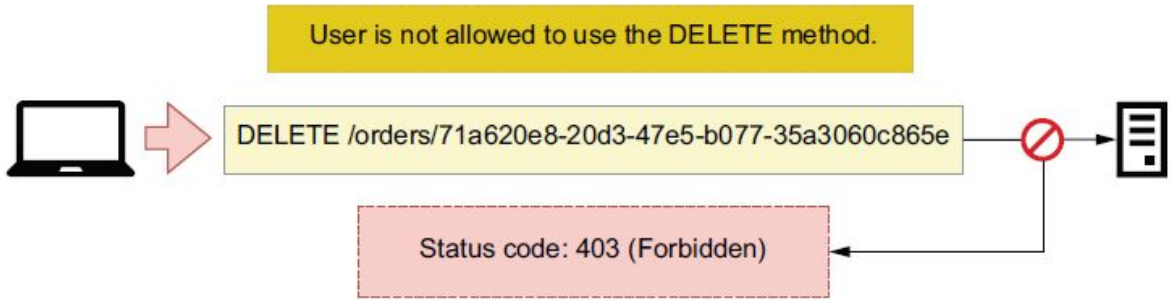
If the requested HTTP method is not available and we don't have a plan to implement it, we respond with the **405 (Method Not Allowed)** status code

Using HTTP status codes to report client errors in the request: User Error-Unauthorized (403)

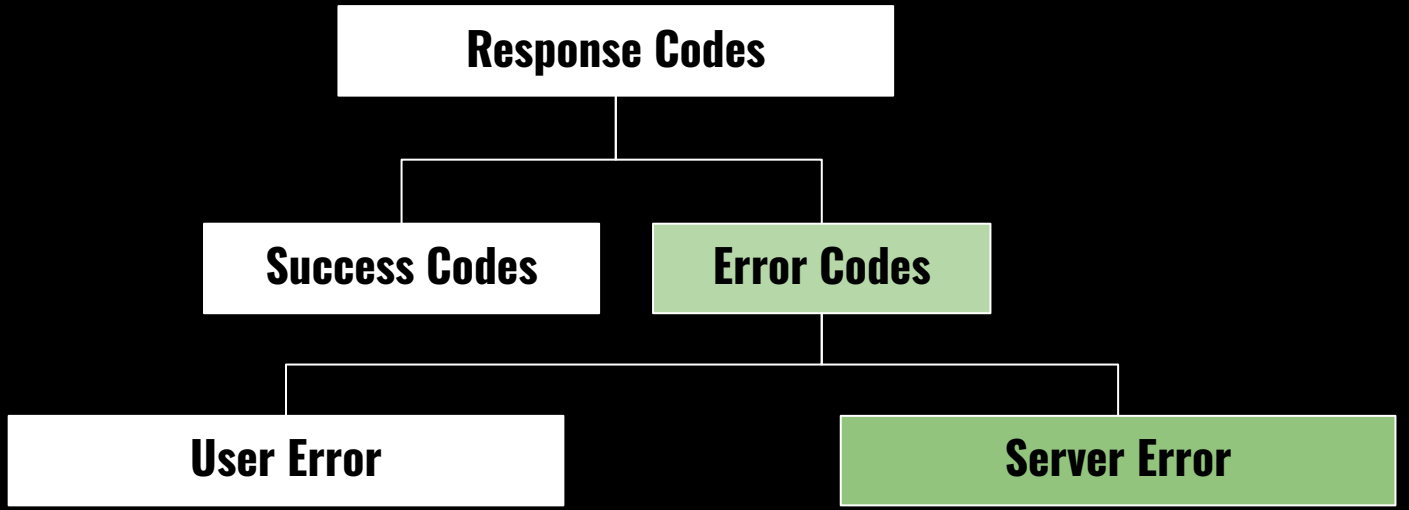
The second error happens when a user is correctly authenticated and tries to use an endpoint or a resource they are not authorized to access

An example is a user trying to access the details of an order that doesn't belong to them

As you can see in figure, we address this scenario with the **403 (Forbidden)** status code, which signals that the user doesn't have permissions to access the requested resource



Using HTTP status codes to create expressive HTTP responses

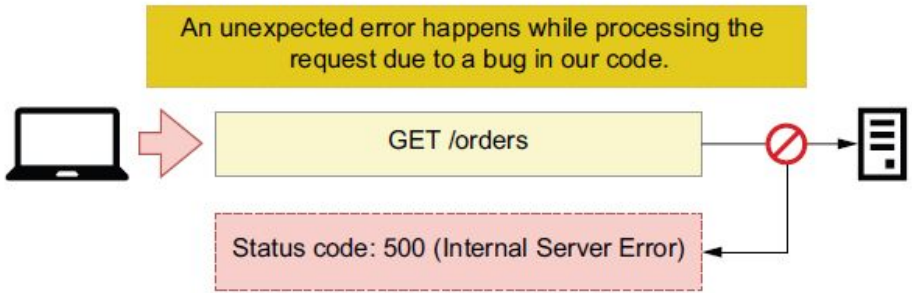


Server Error: Using HTTP status codes to report client errors in the request (500: Internal Server Error)

Errors that are raised in the server due to a bug in our code or to a limitation in our infrastructure

The most common type of error within this category is when our application crashes unexpectedly due to a bug

In those situations, we respond with a **500 (Internal Server Error)** status code, as you can see in figure



Server Error: Using HTTP status codes to report client errors in the request: 503, Server Overloads

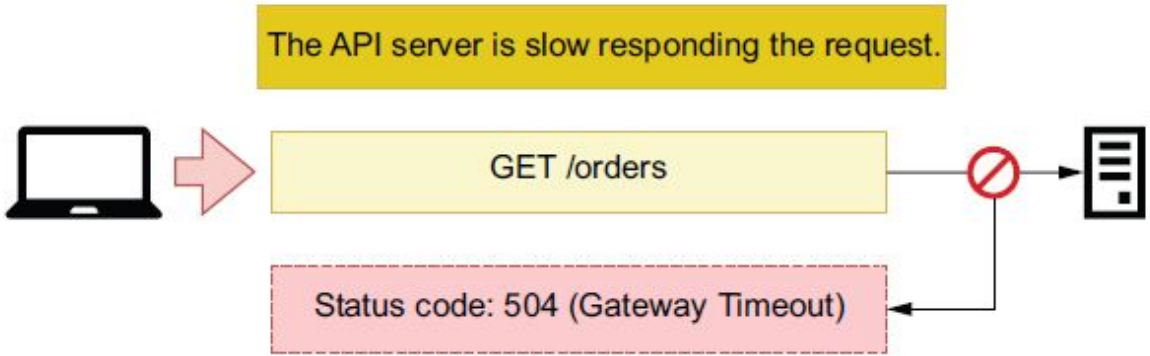
Our API can become unresponsive when the server is overloaded or down for maintenance, and we must let the user know about this by sending an informative status code

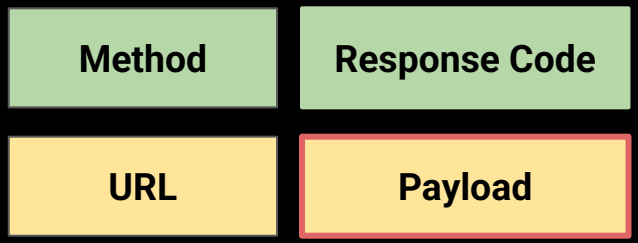
When the server is unable to take on new connections, we must respond with a **503 (Service Unavailable)** status code, which signals that the server is overloaded or down for maintenance and therefore cannot service additional requests

Server Error: Using HTTP status codes to report client errors in the request: 504, Timeout

Our API can become unresponsive when the server is overloaded or down for maintenance, and we must let the user know about this by sending an informative status code

When the server takes too long to respond to the request, we respond with a **504 (Gateway Timeout)** status code





Designing API Payloads

Designing API Payloads

This section explains best practices for designing user-friendly HTTP request and response payloads

Payloads represent the data exchanged between a client and a server through an HTTP request

We send payloads to the server when we want to create or update a resource, and the server sends us payloads when we request data

The usability of an API is very much dependent on good payload design

Poorly designed payloads make APIs difficult to use and result in bad user experiences

It’s therefore important to spend some effort designing high-quality payloads which we are going to do in this section

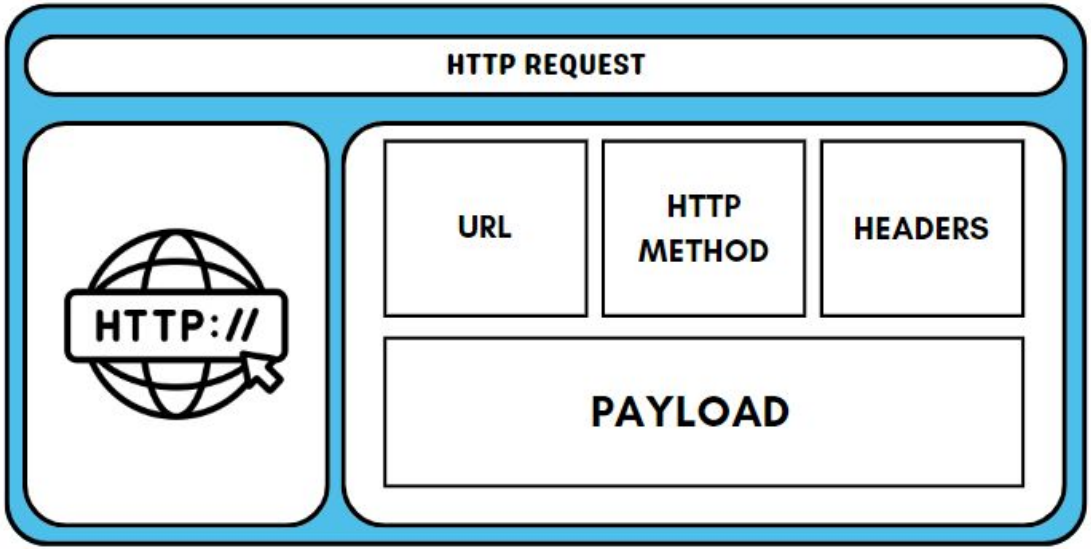
What are HTTP payloads, and when do we use them?

An HTTP request is a message an application client sends to a web server

An HTTP response is the server's reply to that request

HTTP headers include metadata about the request's contents, such as the encoding format

Similarly, an HTTP response includes a status code, a set of headers, and, optionally, a payload



When and When Not To Use Payload

HTTP requests include a payload when we need to send data to the server

For example, a **POST** request typically sends data to create a resource

The HTTP specification allows us to include payloads in all HTTP methods, but it **discourages their use in GET and DELETE methods**

What about responses? Responses may contain a payload depending on the status code

Responses with a **1xx status code**, as well as the **204 (No Content)** status codes, must not include a payload

Let’s learn to design high-quality payloads for all those responses

Response Payload For POST Requests

We use POST requests to create resources

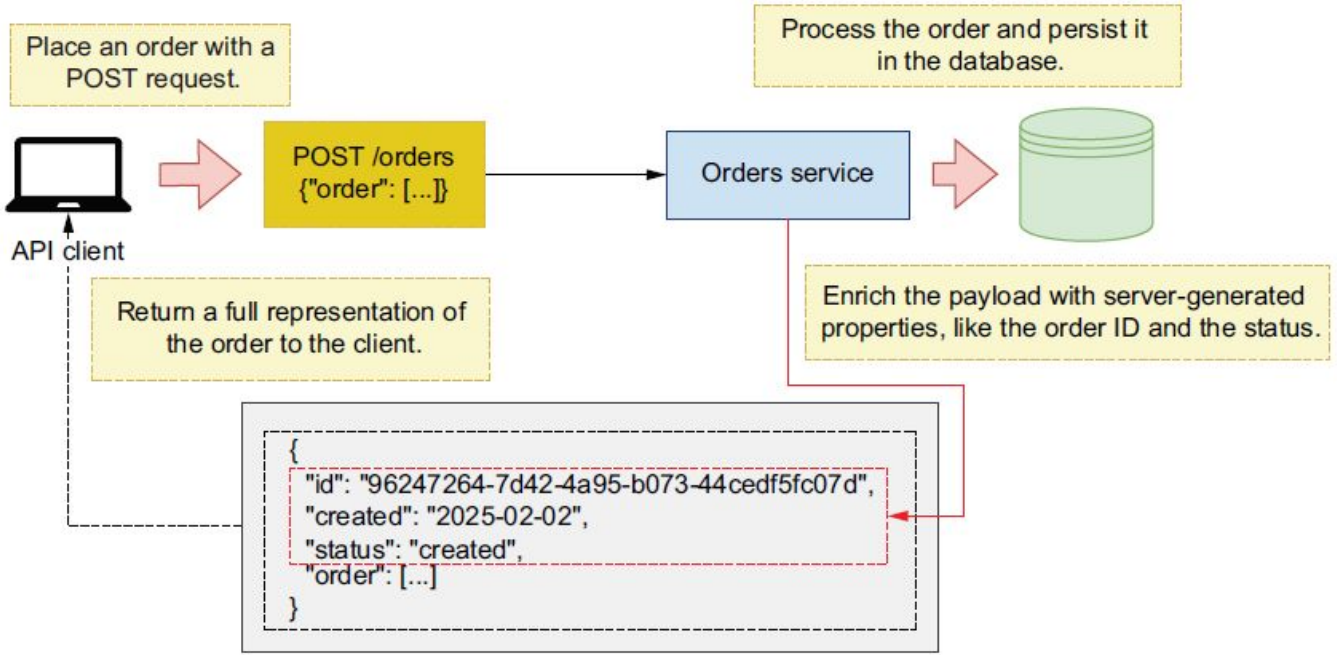
In Swiggy's orders API, we place orders through the POST /orders endpoint

To place an order, we send the list of items we want to buy to the server, which takes responsibility for assigning a unique ID to the order, and therefore the **order's ID must be returned in the response payload**

The server also sets the **time when the order was taken** and its initial status. We call the properties set by the server **server-side or read-only properties**, and we must include them in the response payload

Response Payload For POST Requests

As shown below, it's good practice to return a full representation of the resource in the response to a POST request. This payload serves to validate that the resource was correctly created



Response Payload For GET Requests

We retrieve resources from the server using GET requests

As we established in earlier section, Swiggy's orders API exposes two GET endpoints: the GET /orders and the GET /orders/{orders_id} endpoints

The GET /orders returns a list of orders

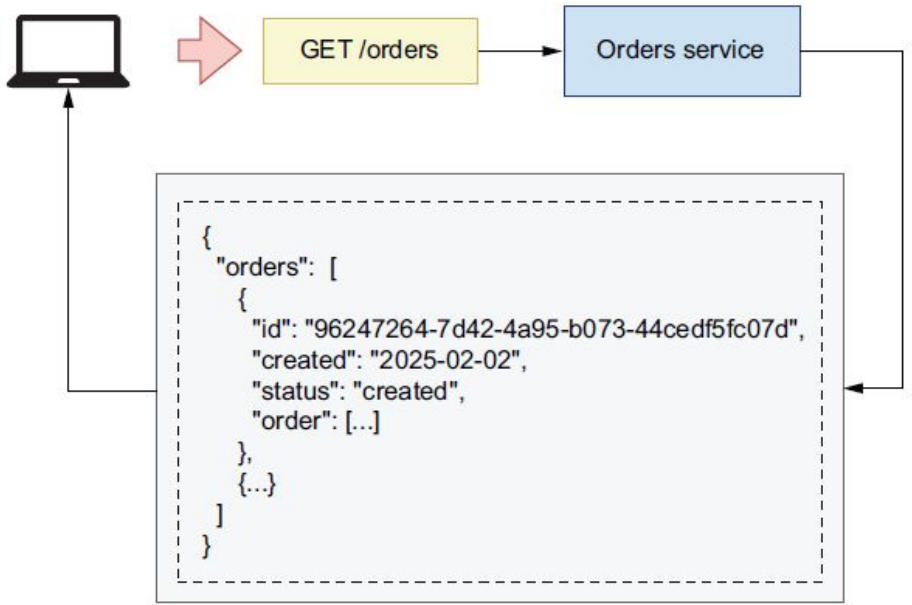
To design the contents of the list, we have two strategies: include a full representation of each order or include a partial representation of each order

Let's check it out in next slide

Response Payload For GET Requests

The first strategy gives the API client **all the information they need in one request**

However, this strategy may compromise the performance of the API when the items in the list are big, resulting in a large response payload

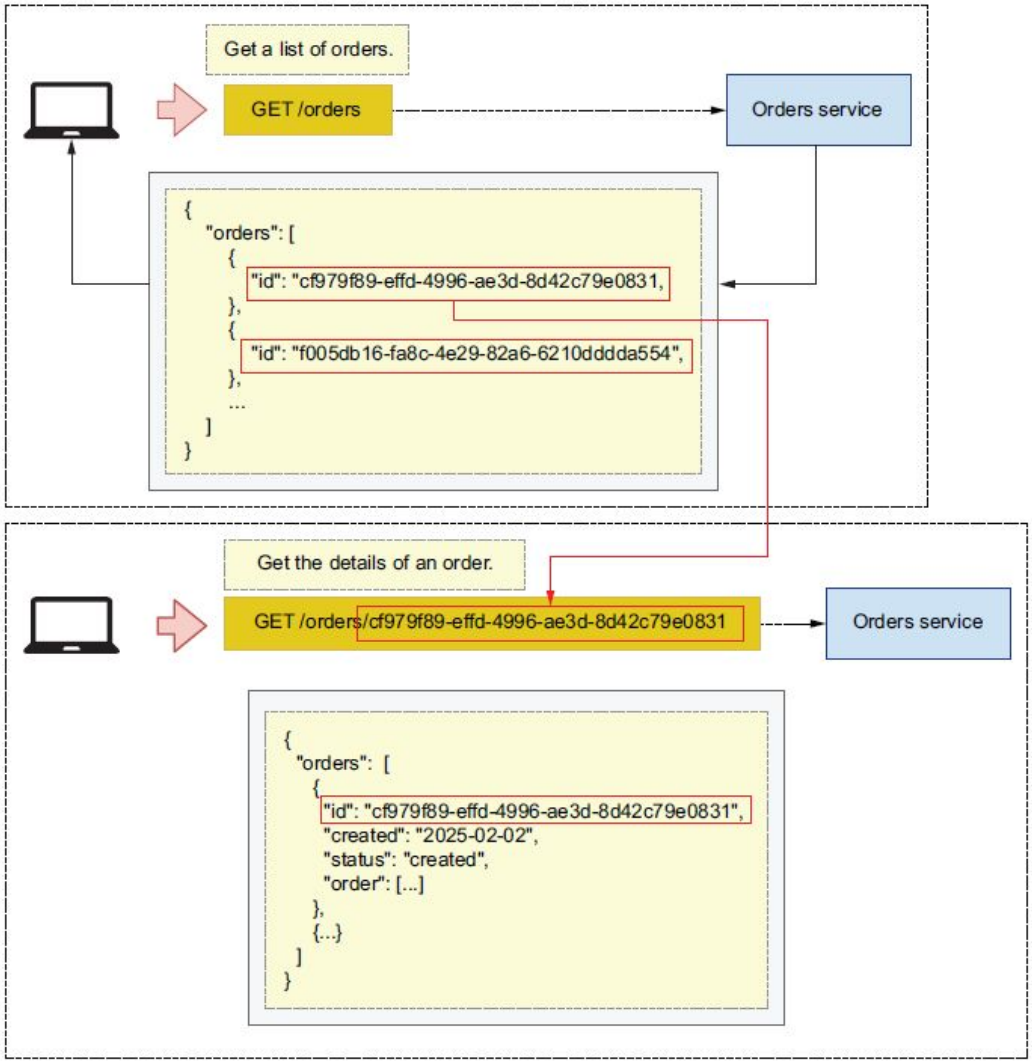


Response Payload For GET Requests

The second strategy for the GET /orders endpoint's payload is to include a **partial representation** of each order

For example, it's common practice to include only the ID of each item in the response of a GET request on a collection endpoint, such as GET /orders

In this situation, the client must call the GET /orders/{order_id} endpoint to get a full representation of each order



Which approach is better?

It depends on the use case

It’s preferable to send a full representation of each resource, especially in public-facing APIs

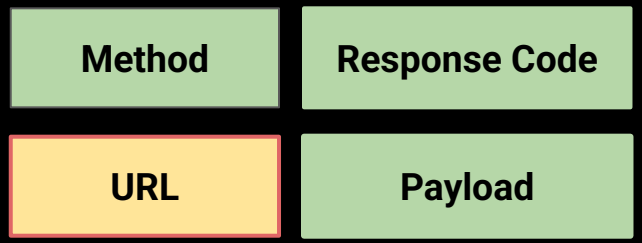
However, if you’re working on an internal API and the full details of each item aren’t needed, you can shorten the payload by including only the properties the client needs

Smaller payloads are faster to process, which results in a better user experience

Finally, **singleton endpoints**, such as the GET /orders/{order_id}, must always **return a full representation** of the resource

Conclusion

Method	Request Payload	Response Payload
GET	No	Yes
PUT	Yes	Yes
PATCH	Yes	Yes
POST	Yes	Yes
DELETE	No	No



Understanding URLs and Query Parameters

URL Overview

URL specifies the location of a resource, for example a web page or a user data in database, an order transaction of Amazon, etc and how it can be retrieved.

Structure of a URL

[Scheme]://[Domain]:[Port]/[Path]?[QueryString]

Scheme is the transport layer: http or https

Domain is your server IP address or DNS if IP is mapped to DNS

Port is applicable.
If DNS is available generally port is also covered in DNS

Path: /login or /signup or /home , etc

Query string: we will cover in couple of slides

http://127.0.0.1:5000/orders

URL Query Parameters

Now let's talk about URL query parameters and how, why, and when you should use them

Some endpoints, such as the GET /orders endpoint of the orders API, return a list of resources

When an endpoint returns a list of resources, it's best practice to allow users to **filter** and **paginate** the results

For example, when using the GET /orders endpoint, we may just want to l**imit the results** to only the five most recent orders or **only cancelled orders**

URL query parameters allow us to accomplish those goals by enabling filter like conditions on resources

URL Query Parameters

URL query parameters are key-value pairs that form part of a URL but are separated from the URL path by a question mark

For example, if we want to call the GET /orders endpoint and filter the results by cancelled orders, we may write something like this

```
GET /orders?cancelled=true
```

We can chain multiple query parameters within the same URL by separating them with ampersands

To filter the GET /orders endpoint by cancelled orders and restrict the number of results to 5, we make the following API request

```
GET /orders?cancelled=true&Limit=5
```

URL Query Parameters

It's also common practice to allow API clients to paginate results

Pagination consists of slicing the results into different sets and serving one set at a time

We can use several strategies to paginate results, but the most common approach is using a ***page*** and a ***per_page*** combination of parameters

page represents a set of the data, while ***per_page*** tells us how many items we want to include in each set

The server uses per_page's value to determine how many sets of the data we'll get. We combine both parameters in an API request as in the following example

```
GET /orders?page=1&per_page=10
```

Hands On - Query Parameter

Hands On - Caching Using Redis