Cloud Computing and SE

Assignment 1

Dr. Daniel Yellin

Cloud Computing and SE

Assignment 1

Dr. Daniel Yellin

Assignment 1: a RESTful API

- In the assignment, your application will need to:
  - Invoke a RESTful API (Topic 2)
  - Provide a RESTful API (Topic 2)
  - Use Docker containers to package your application and submit you assignment (Topic 3)

- The assignment can be done alone or in a group of two/three students together (no more than 3 in one group).

- Eventhough we have not covered Docker Containers yet (we will next lecture), most of the assignment is programming the REST server (Topic 2) which we already covered. You should get started now. The assignment will be due two weeks after the next lecture on Docker. I will notify you shortly on the date of that lecture as it will not be at the regular time, due to holidays which fall out on Mondays.

Testing your program

- To test your assignment, we will
  - Build your docker image using the source files and the Dockerfile you provide
  - Run your docker image (container)
  - Invoke APIs on the program at the specified port

- We will check whether the outputs from each request are correct
  - We will check both the returned content and the returned status code
  - Sometimes we expect the status to return an error code
- We may compare source code of different submissions to check for originality (no copying!)

Assignment 1: a RESTful API for meals

The RESTful API will allow users to:

- create and store *dishes*. A dish is given by its name:
    - “peanut butter and jelly sandwich”
    - “chicken soup”
    - “tossed salad”
- create and store *meals* by specifying the 3 dishes that comprise that meal (appetizer, main*, dessert)
- retrieve, update, and delete dishes and meals

* We use “main” instead of “entrée” because in some countries (France), “entrée” is the appetizer

Computing additional information for dishes

- When the user adds a *dish*, the program will compute
  - the number of *calories*,
  - the *serving size* (in grams),
  - the amount of *sodium* (in mg), and
  - the amount of *sugar* (in grams)

for that dish. It will invoke a the REST API *http://api.api-ninjas.com/v1/nutrition* to compute this information. (See slides 44-46)

The dish and dishes resources

/dishes

This is a collection class, containing all the dishes

/dishes/{ID} or /dishes/{name}

Each dish resource is expressed by the following JSON object:

```
{
"name": <string>,
"ID": <number>,  # must be an integer
"cal": <number>,
"size": <number>,
"sodium": <number>,
"sugar": <number>
}
```

Example:

```
{
"name": "focaccia",
"ID": 2,
"cal": 251.9,
"size": 100.0,
"sodium": 570,
"sugar": 1.8
}
```

Naming

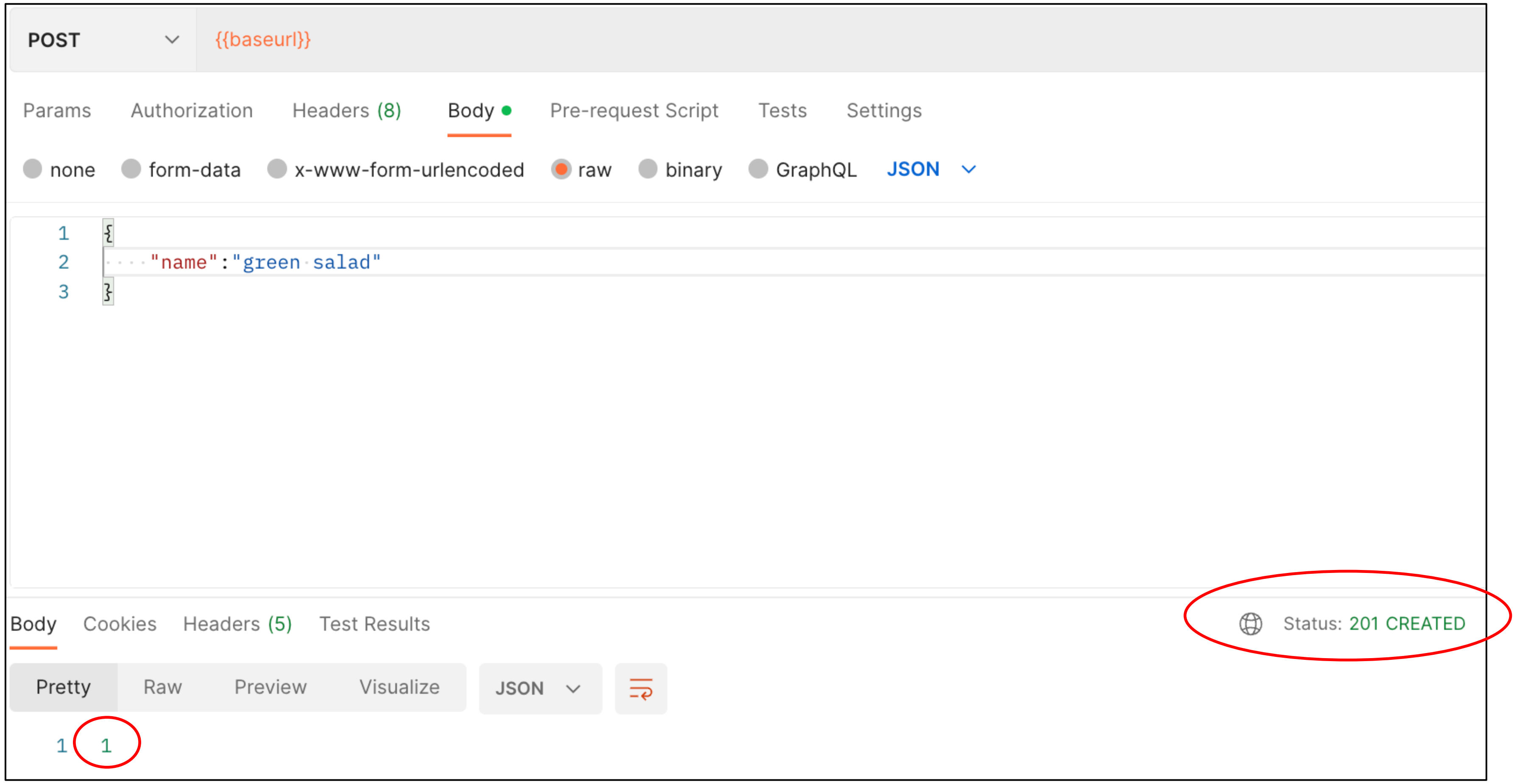
- The order of the fields in a JSON object is not important. *Any* order is ok.
- The field must be spelled *exactly*.
    - “Focacia” is *different* from “focacia” and from “ Focacia”
    - “ID” is different from “Id” and from “id”.
    - “apple pie” is *different* from “apple-pie” and from “Apple Pie” and from ”apples pie”.
- White spaces make a difference:
    - “salad” is *different* from “ salad” and from “salad ” (leading or trailing whitespaces).
    - “green salad” is *different* from “green salad” (two spaces instead of one in between “green” and “salad”).

Requests on the /dishes resource

/dishes

- **POST** will add a dish of the given name.   If successful, it returns the dish ID, a positive integer, and the code 201 (Resource successfully created).
- **POST** may also return a non-positive ID with the following meaning:
  - *0 means that request content-type is not application/json. Status code 415 (Unsupported Media Type)*
  - *-1 means that 'name' parameter was not specified in the message body. Status code 422 (Unprocessable Content)*
  - *-2 means that dish of given name already exists. Status code 422 (Unprocessable Content)*
  - *-3 means that api.api-ninjas.com/v1/nutrition does not recognize this dish name. Status code 422 (Unprocessable Content)*
  - *-4 means that api.api-ninjas.com/v1/nutrition was not reachable. Status code 504 (Gateway Timeout)*
- **GET** will return the JSON object listing all dishes, indexed by ID

POST /dishes


POST {{baseurl}}

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "name":"green salad"
}
```

Status: 201 CREATED

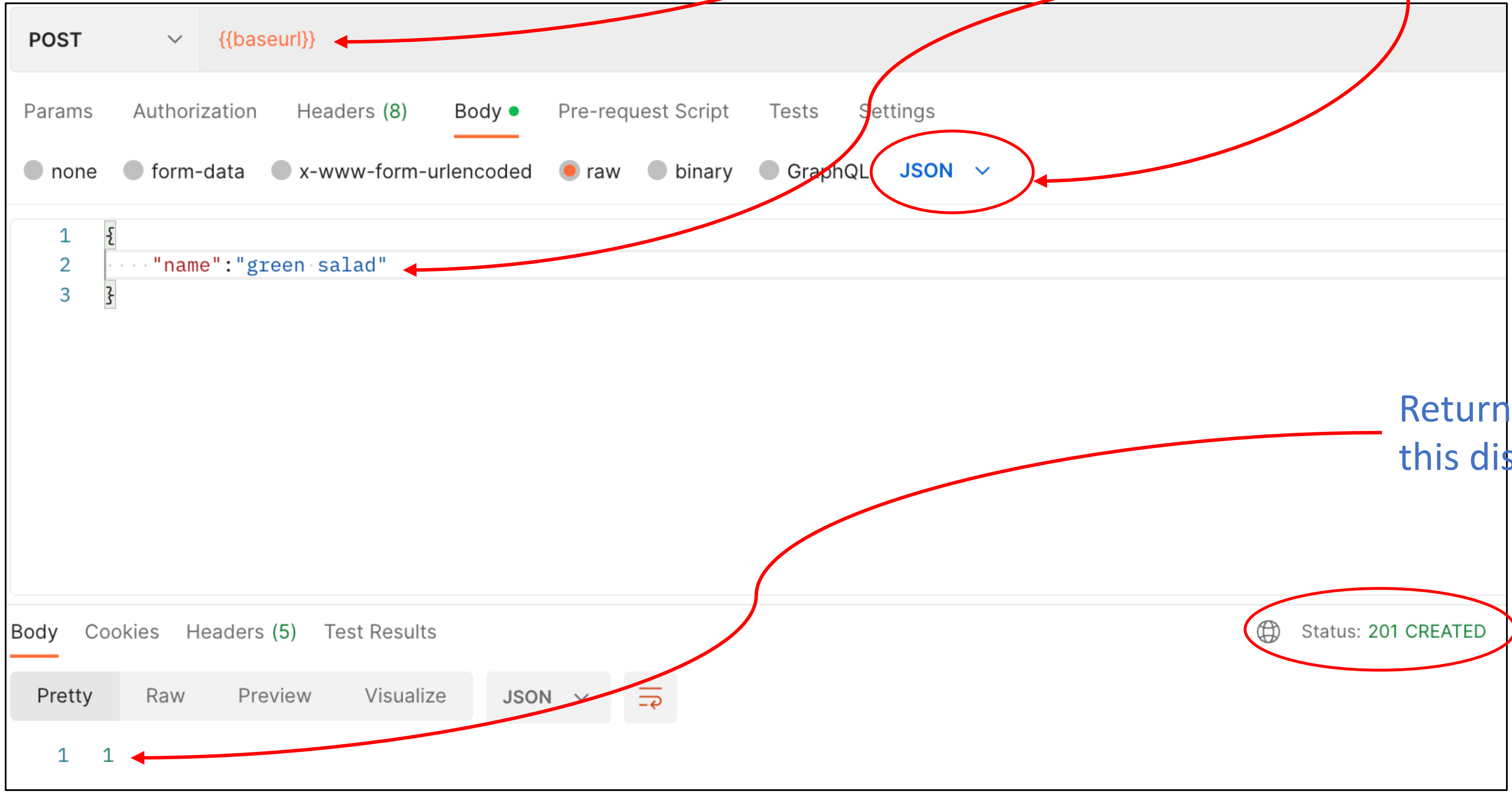
```
1
```

Dish successfully created (response code 201). Returned ID = 1 for dish.

POST /dishes

```
curl --location 'http://0.0.0.0:80/dishes' \
--header 'Content-Type: application/json' \
--data '{
"name":"green salad"
}'
```

curl request



```
POST    {{baseurl}}
```

```
{
····"name":"green·salad"
}
```

Status: 201 CREATED

```
1
```

Returned ID for this dish

Postman

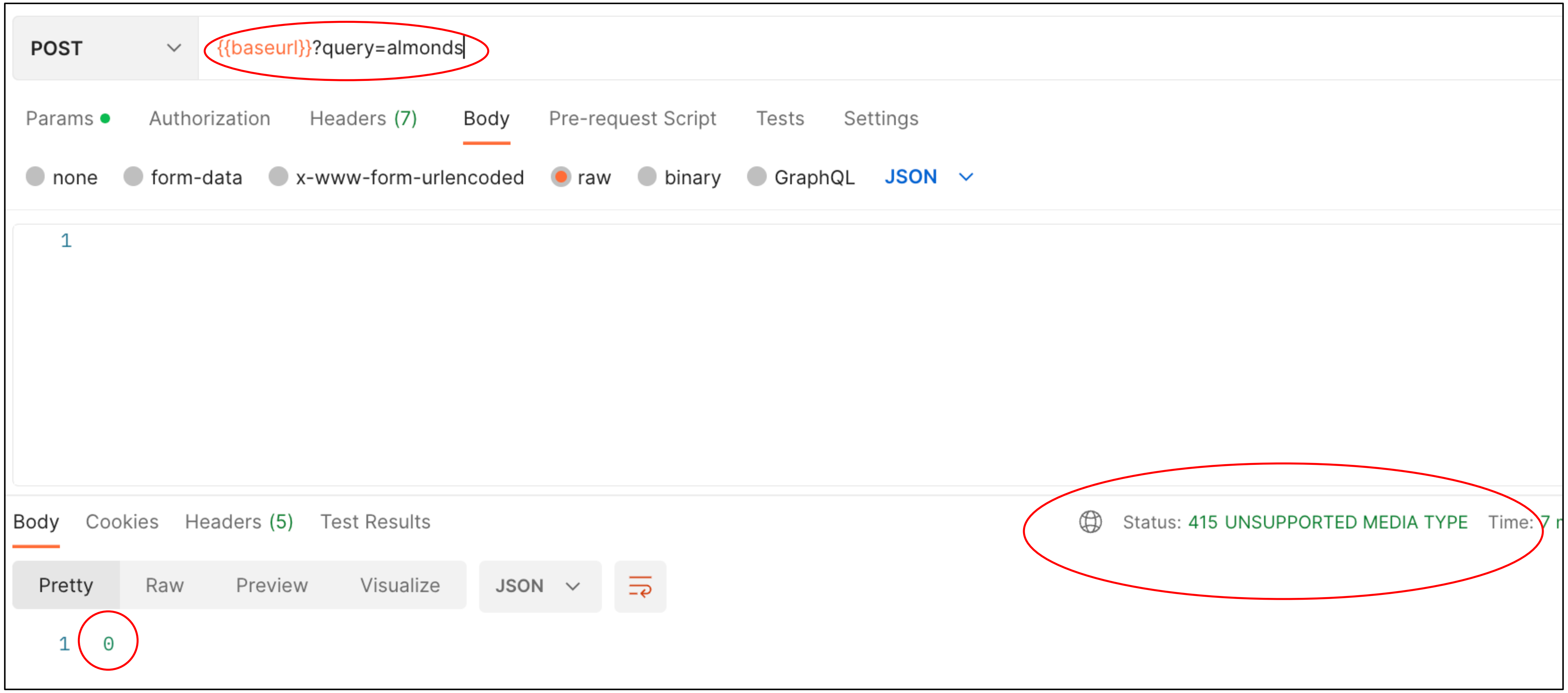
POST using python requests package

- Header of each request must include ‘Content-Type: application/json’
- On the right is the request using the Python requests package.

```
import requests
import json

url = "http://0.0.0.0:80/dishes"

payload = json.dumps({"name": "green salad"})

headers = {'Content-Type': 'application/json’}

response = requests.request("POST", url,
            headers=headers, data=payload)

print(response.text)
```

Response to non-JSON POST request

```
curl --location --request POST 'http://0.0.0.0:80/dishes?query=almonds' \
--data ''
```



POST {{baseurl}}?query=almonds

Params ● Authorization Headers (7) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
1
```

Body Cookies Headers (5) Test Results

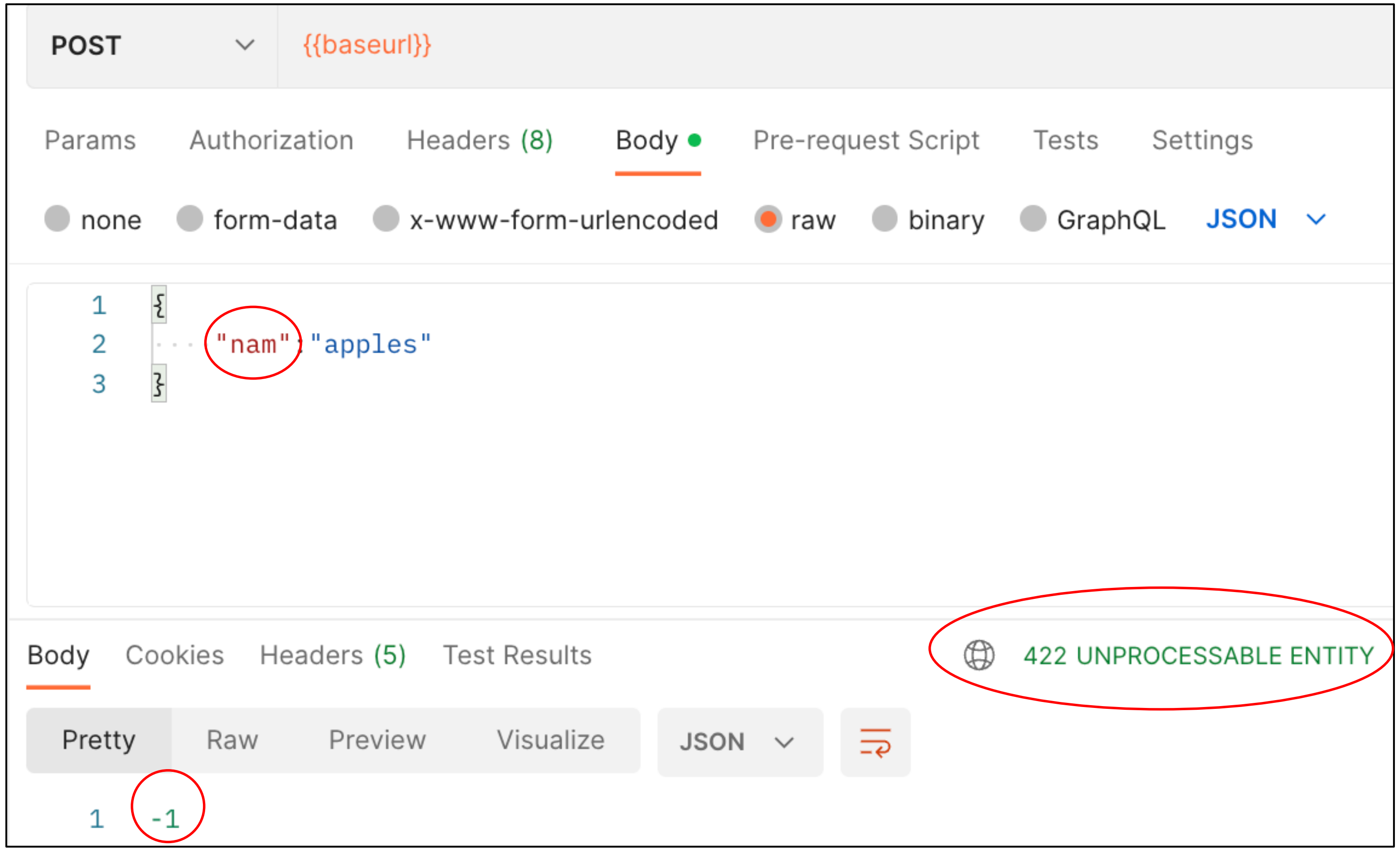
Status: 415 UNSUPPORTED MEDIA TYPE Time: 7

Pretty Raw Preview Visualize JSON

```
0
```

Sent parameter in query string instead of JSON object. Return ID = 0 and status code 415.

Response to badly formatted JSON request


POST {{baseurl}}

Params | Authorization | Headers (8) | Body ● | Pre-request Script | Tests | Settings

none | form-data | x-www-form-urlencoded | raw | binary | GraphQL | JSON

```
{
    "nam": "apples"
}
```

Body | Cookies | Headers (5) | Test Results

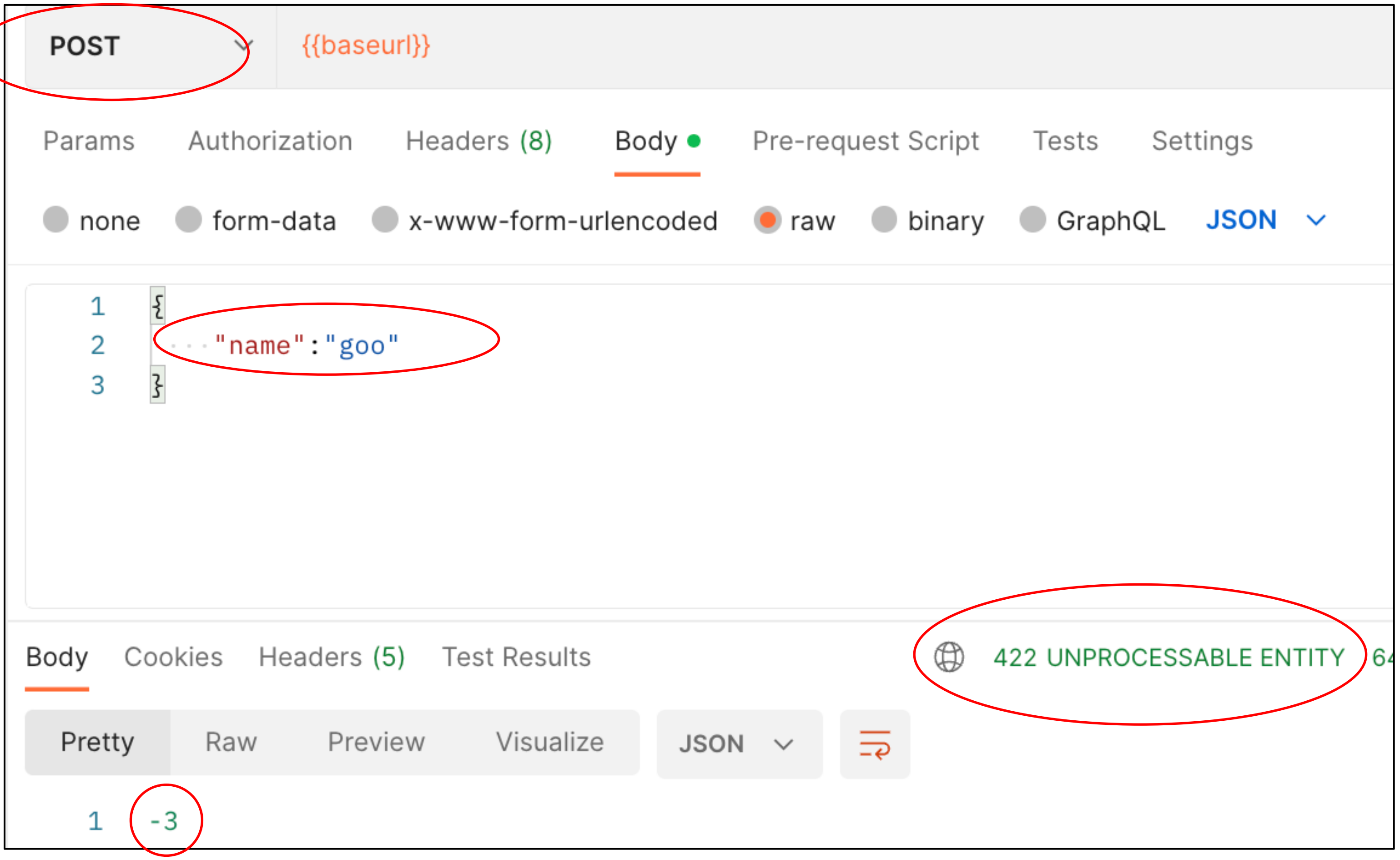
422 UNPROCESSABLE ENTITY

Pretty | Raw | Preview | Visualize | JSON

```
-1
```

Expected JSON field "name" but was sent JSON field "nam". Return ID = -1 and status code 400.

POST a dish that API-NINJA does not recognize


POST {{baseurl}}

Params · Authorization · Headers (8) · Body · Pre-request Script · Tests · Settings

none · form-data · x-www-form-urlencoded · [✓ raw] · binary · GraphQL · JSON

```
{
    "name":"goo"
}
```

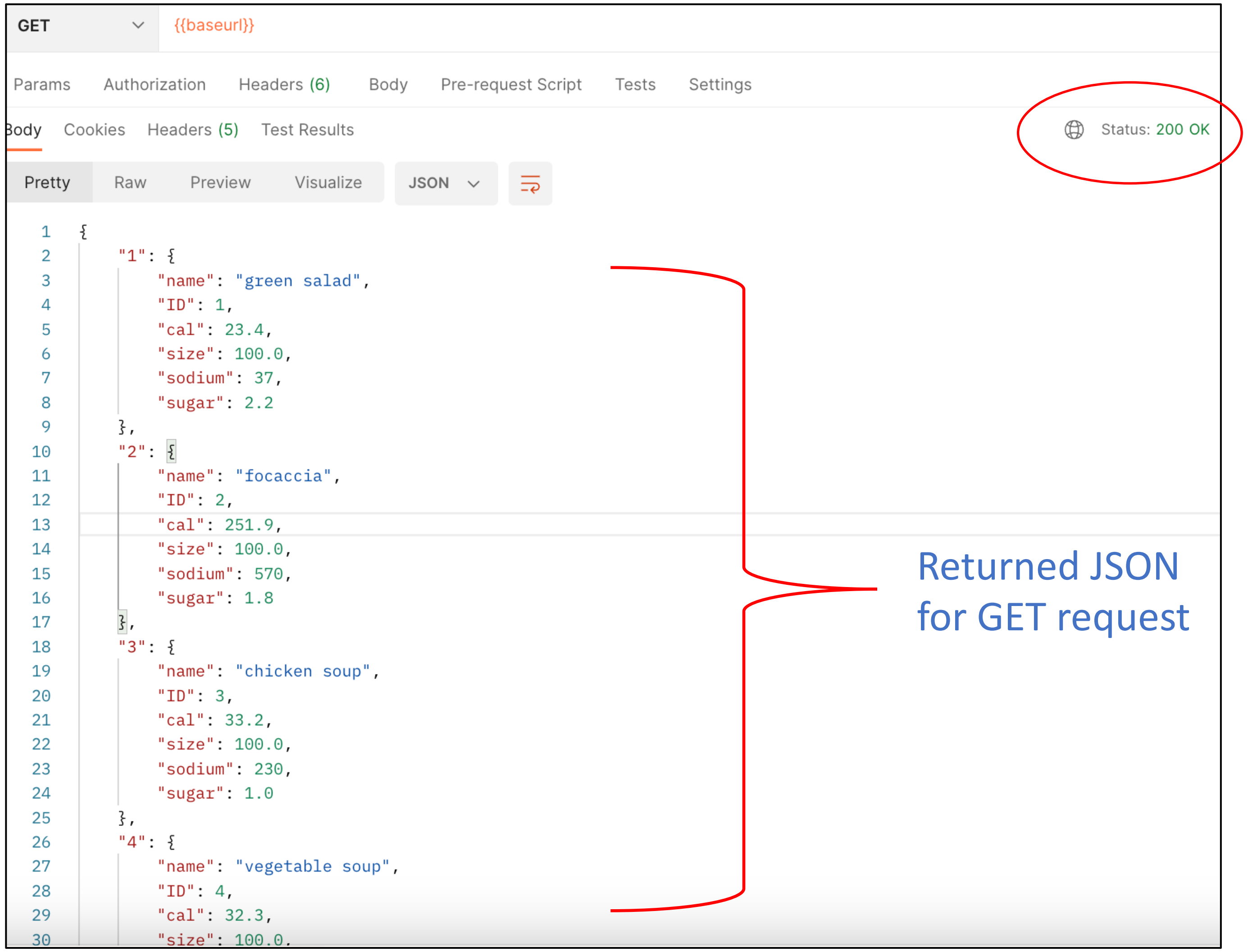
Body · Cookies · Headers (5) · Test Results · 422 UNPROCESSABLE ENTITY

Pretty · Raw · Preview · Visualize · JSON

```
-3
```

API-NINJA does not recognize the dish called "goo". Return ID = -3 and status code 400.

GET /dishes



GET {{baseurl}}

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

```
{
    "1": {
        "name": "green salad",
        "ID": 1,
        "cal": 23.4,
        "size": 100.0,
        "sodium": 37,
        "sugar": 2.2
    },
    "2": {
        "name": "focaccia",
        "ID": 2,
        "cal": 251.9,
        "size": 100.0,
        "sodium": 570,
        "sugar": 1.8
    },
    "3": {
        "name": "chicken soup",
        "ID": 3,
        "cal": 33.2,
        "size": 100.0,
        "sodium": 230,
        "sugar": 1.0
    },
    "4": {
        "name": "vegetable soup",
        "ID": 4,
        "cal": 32.3,
        "size": 100.0,
```

Returned JSON for GET request

```
curl --location
'http://0.0.0.0:80/dishes' \
--data '
```

curl request

```
import requests

url = "http://0.0.0.0:80/dishes"

payload = ""
headers = {}

response = requests.request("GET", url,
          headers=headers, data=payload)

print(response.text)
```

Python request

DELETE /dishes

- Deleting the entire collection is not allowed.
- This request will automatically return error message “This method is not allowed for the requested URL” with status code 405 (Method not allowed) since this method is not implemented in your code.

Requests on the /dishes/dish resource

There are two ways to specify a dish – its ID or its name

/dishes/{ID}

- **GET** sends the ID of the dish and will receives back the JSON object consisting of the name, ID, calories, sodium and sugar content of the resource
- **DELETE** sends the ID of the dish and removes that dish from /dishes. It returns the ID of the deleted dish.

/dishes/{name}

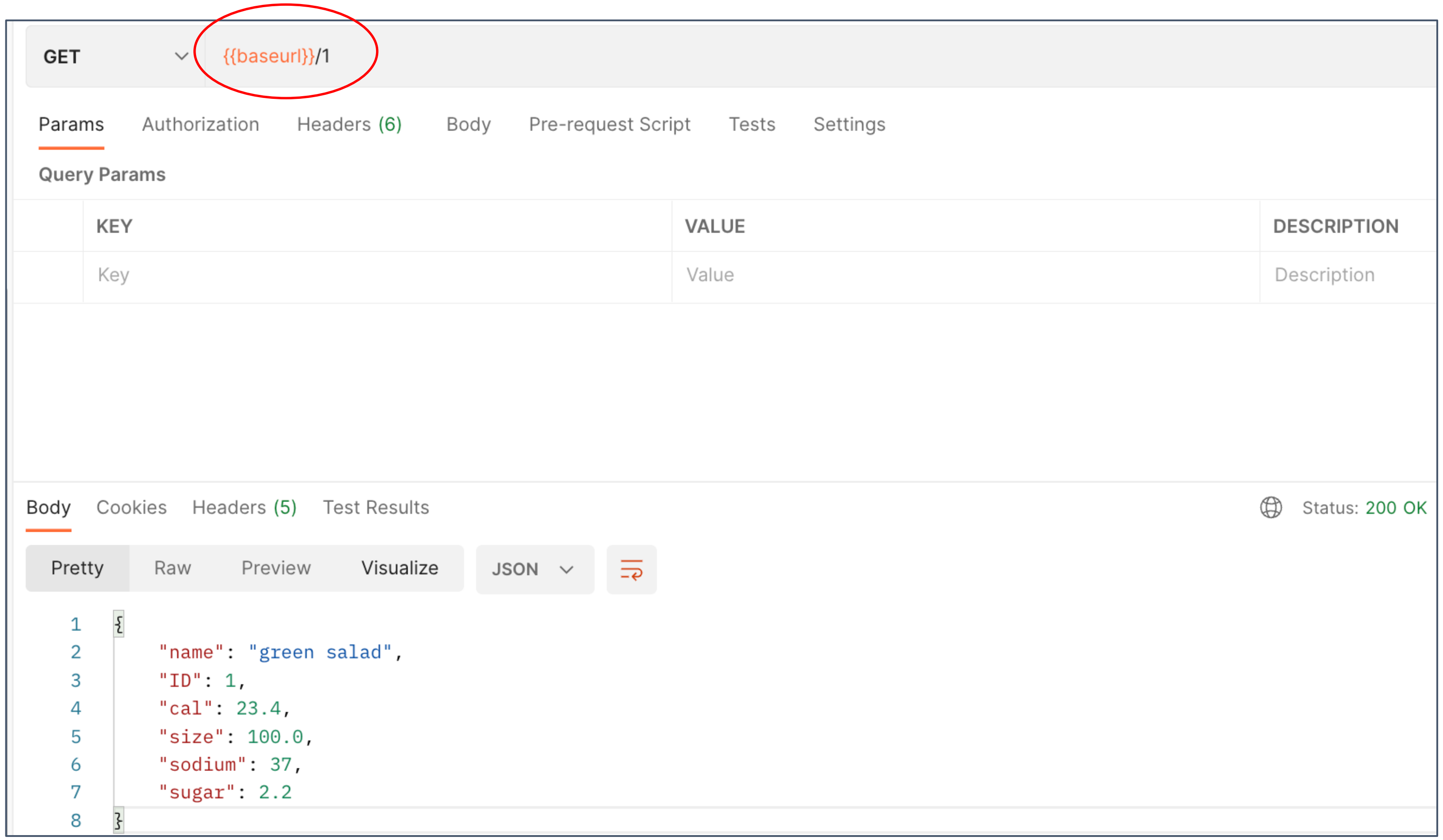
- **GET** and **DELETE** are the same as the above except instead of sending the ID of the dish, one sends the name of the dish . NOTE that for DELETE, this endpoint still returns the ID of the dish (not the name).

- No other operations (e.g., PUT) on these resources are permitted as the name of a dish cannot change, and the other resources are computed from the name

GET and DELETE error responses

- If dish name or dish ID does not exist, the GET or DELETE request returns the response -5 with status code 404 (Not Found)
  - In this case, the URI /dishes/{ID} or /dishes/{name} is not found.

GET /dishes/{ID}


GET {{baseurl}}/1

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK

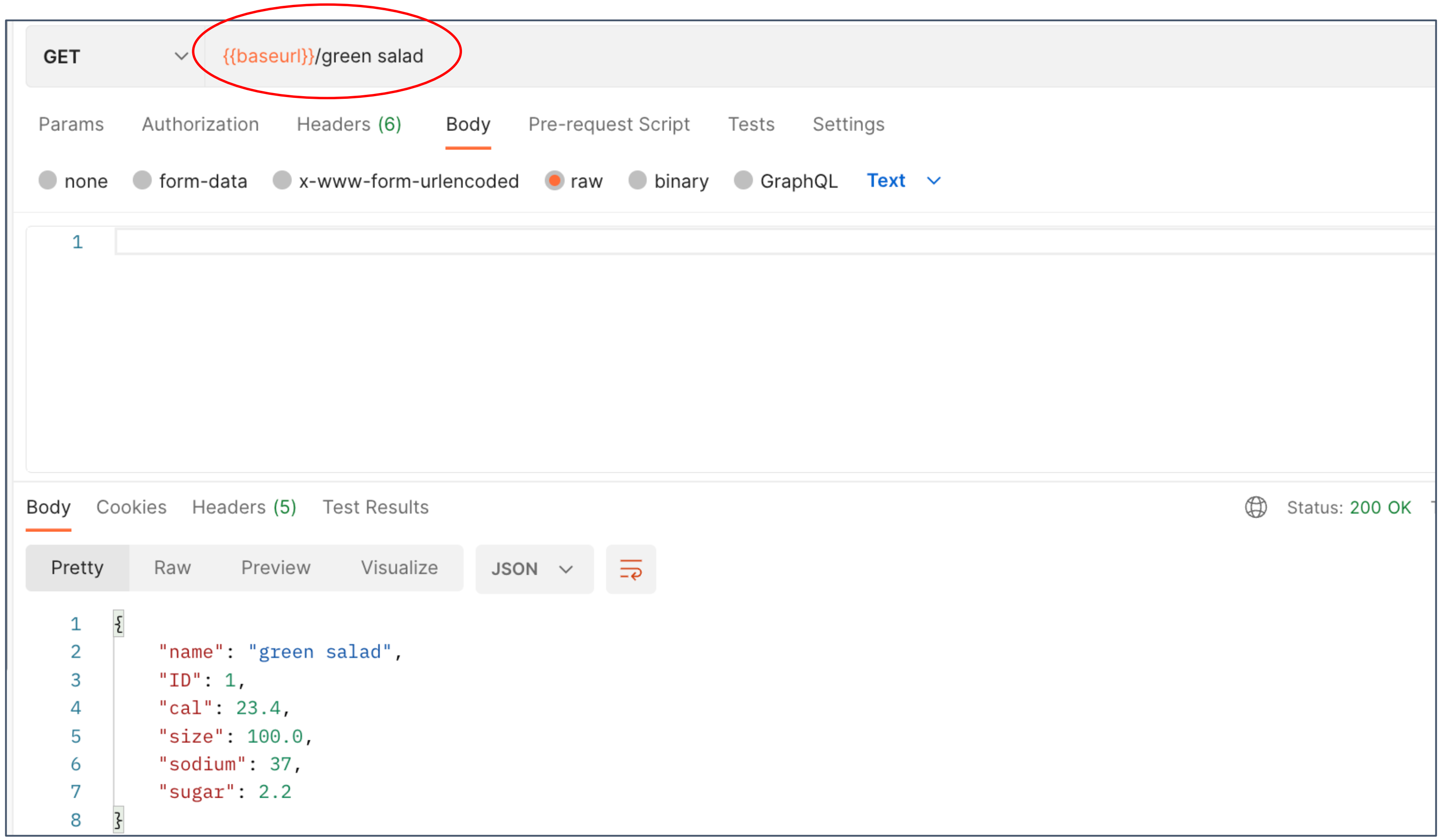
```
{
    "name": "green salad",
    "ID": 1,
    "cal": 23.4,
    "size": 100.0,
    "sodium": 37,
    "sugar": 2.2
}
```

```python
import requests

url = "http://0.0.0.0:80/dishes/1"

payload={}
headers = {}

response = requests.request("GET", url,
          headers=headers, data=payload)

print(response.text)
```

Python request

Returns the JSON object for dish with ID = 1 with status code 200.

GET /dishes/{name}



GET {{baseurl}}/green salad

Params · Authorization · Headers (6) · Body · Pre-request Script · Tests · Settings

none · form-data · x-www-form-urlencoded · [✓ raw] · binary · GraphQL · Text

Body · Cookies · Headers (5) · Test Results — Status: 200 OK

Pretty · Raw · Preview · Visualize · JSON

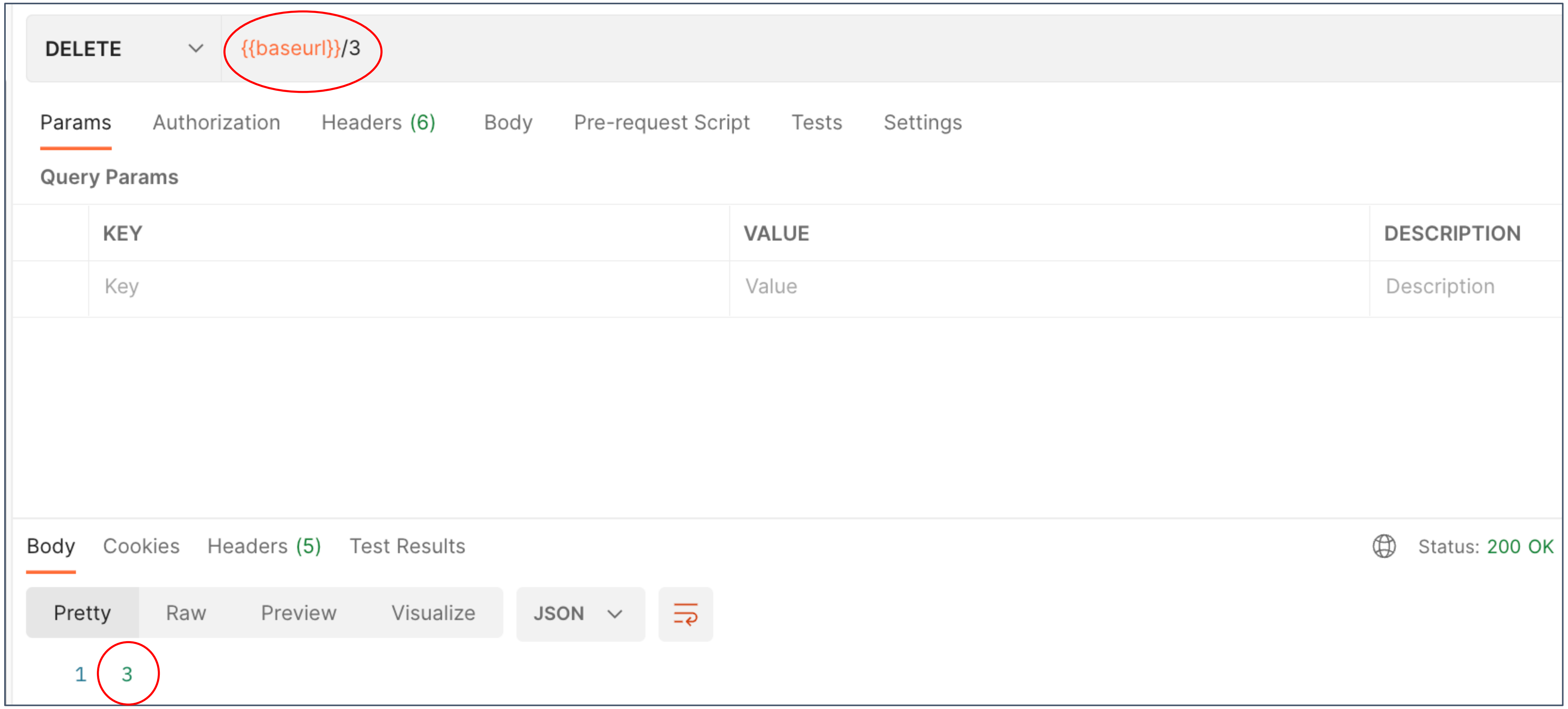
```
{
    "name": "green salad",
    "ID": 1,
    "cal": 23.4,
    "size": 100.0,
    "sodium": 37,
    "sugar": 2.2
}
```

```
import requests

url = "http://0.0.0.0:80/dishes/green salad"

payload = ""
headers = {}

response = requests.request("GET", url,
        headers=headers, data=payload)

print(response.text)
```

Python request

Returns the JSON object for dish with "name" = "green salad" with return status 200.

DELETE /dishes/{ID}



DELETE {{baseurl}}/3

Params | Authorization | Headers (6) | Body | Pre-request Script | Tests | Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

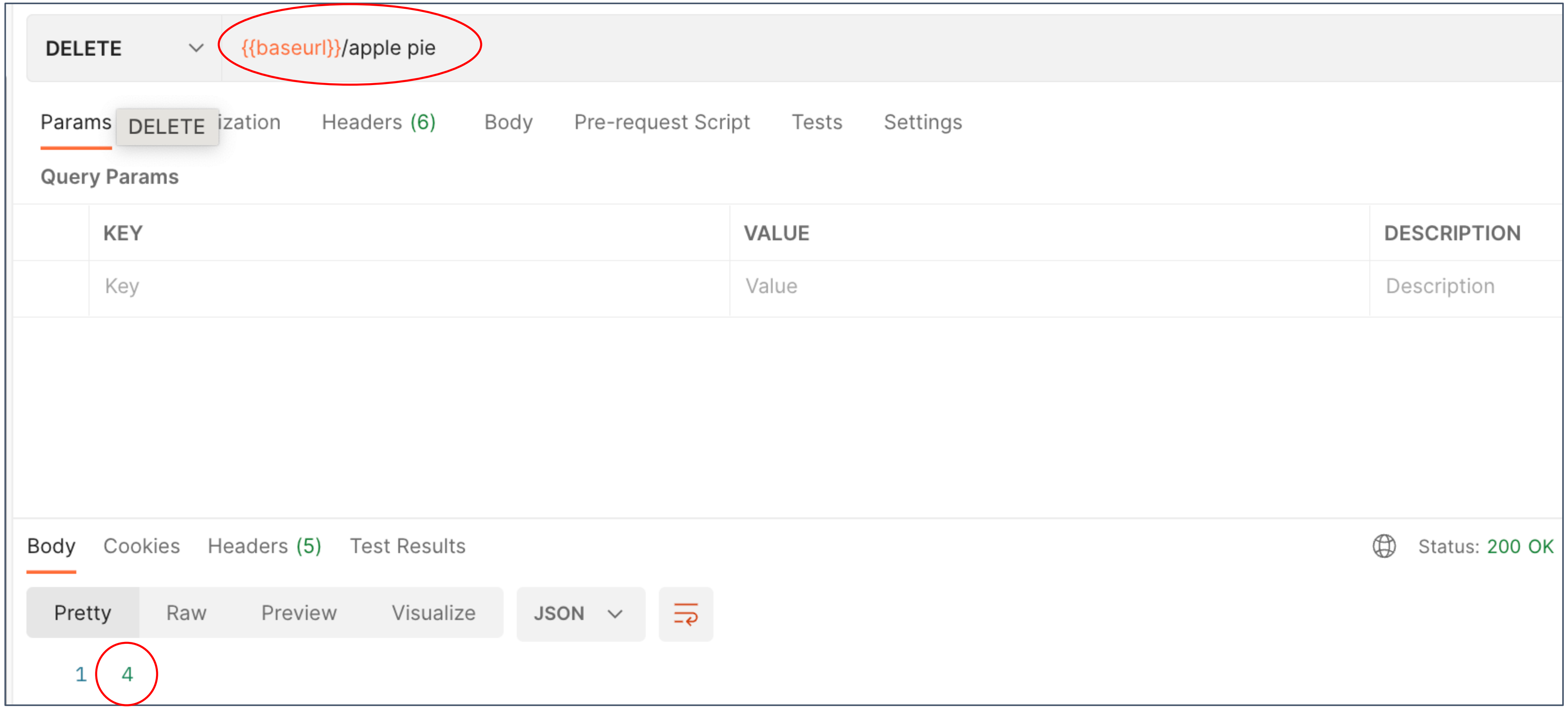
Body | Cookies | Headers (5) | Test Results — Status: 200 OK

Pretty | Raw | Preview | Visualize | JSON

```
3
```

Deletes the dish with ID = 3 with status code 200 (successfully deleted the resource)

DELETE /dishes/{name}



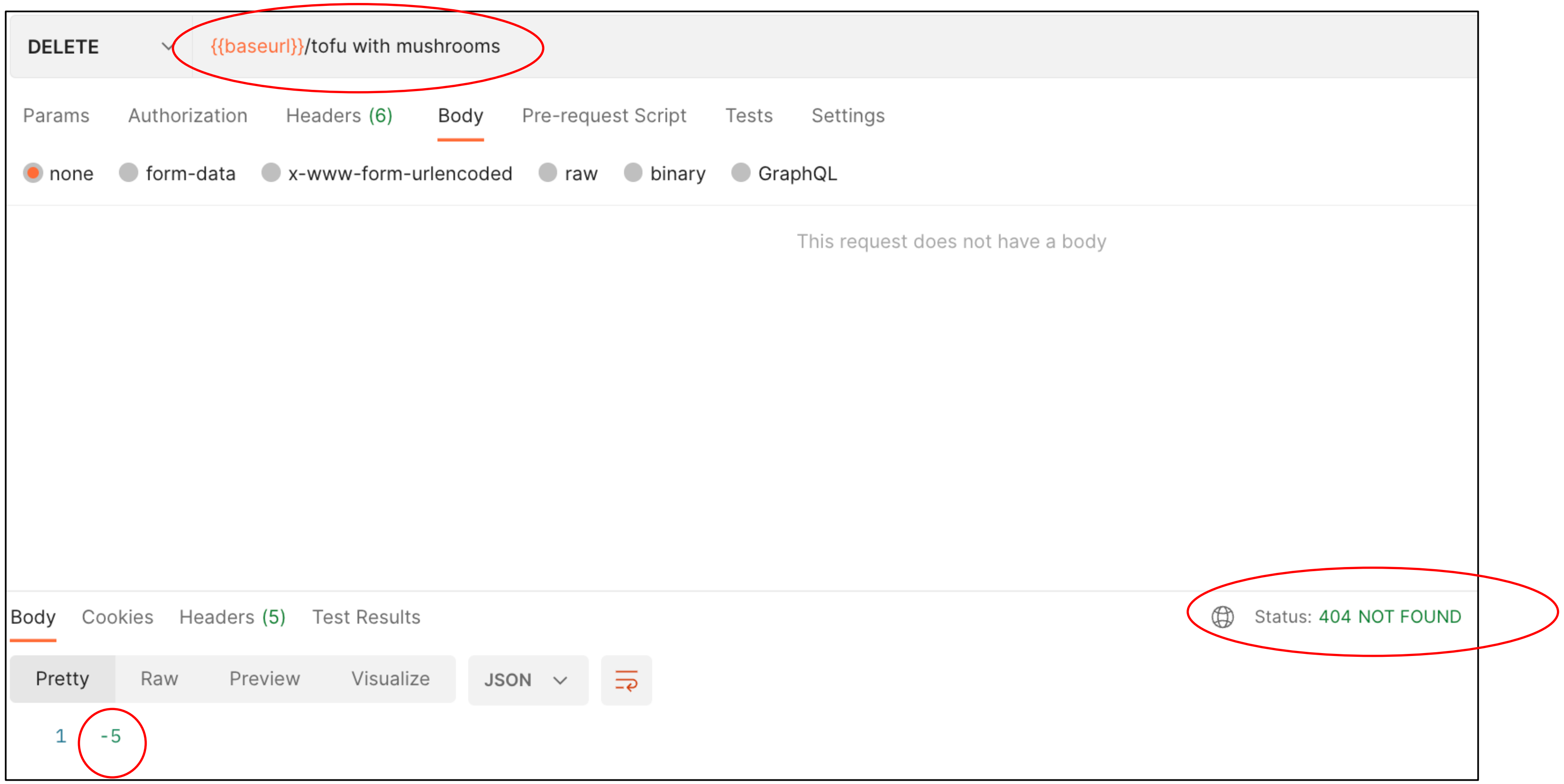
{{baseurl}}/apple pie

Status: 200 OK

```
4
```

Deletes the dish with name = “apple pie” with status 200 and returns the deleted item’s ID

DELETE /dishes/{name} with incorrect name


DELETE {{baseurl}}/tofu with mushrooms

[✓ none] form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Status: 404 NOT FOUND

```
-5
```

The dish with name = “tofu with mushrooms” does not exist. Return ID = -5 and status code 404.

The meal and meals resources

Note that a meal does not have a serving size like dishes do.

/meals

This is a collection class, containing all the meals

/meals/meal

Each meal resource is expressed by the following JSON object:

```
{
“name”: <string>,           # name of meal
“ID”: <number>,             # ID of meal
“appetizer”: <number>,  # ID of appetizer dish
“main”: <number>,           # ID of main dish
“dessert”: <number>,        # ID of dessert dish
“cal”: <number>,            # total number of calories in meal
“sodium”: <number>          # total amount of sodium in meal
“sugar”: <number>           # total amount of sugar in meal
}
```

Example:

```
{
"name": "chicken special",
"ID": 2,
"appetizer": 3,
"main": 16,
"dessert": 15,
"cal": 812.4,
"sodium": 1018,
"sugar": 1.9000000000000001
}
```

Requests on the /meals resource

/meals

- **POST** will send a JSON request object with the meal name and dish IDs (appetizer, main, dessert). It will add that meal and return ID of the meal. If successful it will return a response code 201.
- **POST** may also return a non-positive ID with the following meaning:
  - *0 means that request content-type is not application/json. Status code 415 (Unsupported Media Type)*
  - *-1 means that one of the required parameters was not given or not specified correctly. Status code 422 (Unprocessable Entity)*
  - *-2 means that a meal of the given name already exists. Status code 422 (Unprocessable Entity)*
  - *-6 means that one of the sent dish IDs (appetizer, main, dessert) does not exist. Status code 422 (Unprocessable Entity)**
  - **GET** will return the JSON object listing all meals, indexed by ID

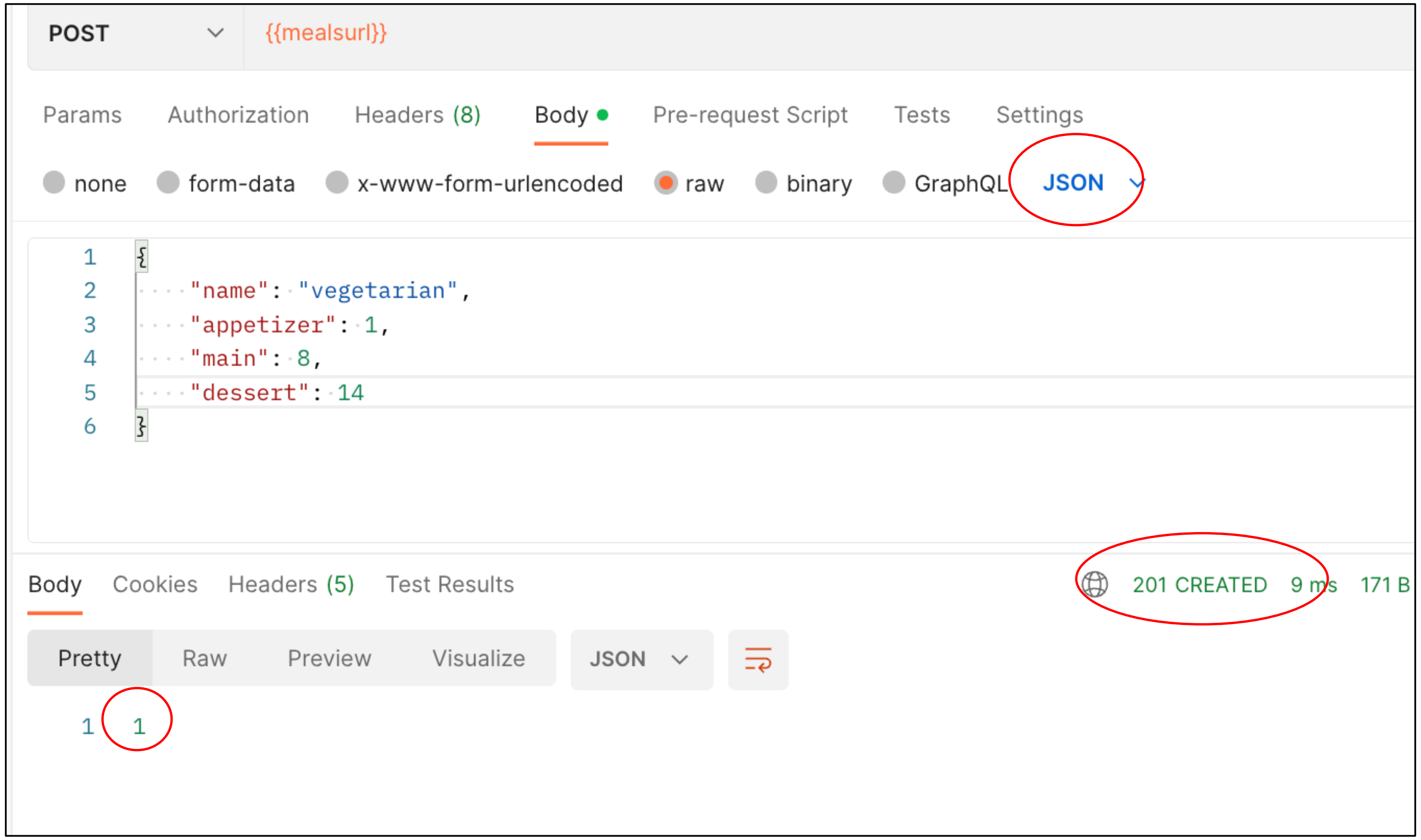
* This does not return an Error Code 404 (Not Found) because the URI /meals is found. It is a parameter in the payload which is not found.

Computing additional information for meals

When the user creates a *meal*, the program will compute the *total calories, sodium, and sugar* for that meal by summing the calories,sodium and sugar of the individual dishes that make up the meal.

- The calories, sodium, and sugar quantities for the mean needs to be updated if one of the dishes of the meal changes

POST /meals



POST {{mealsurl}}

Params · Authorization · Headers (8) · Body ● · Pre-request Script · Tests · Settings

none · form-data · x-www-form-urlencoded · [✓ raw] · binary · GraphQL · JSON

```
{
    "name": "vegetarian",
    "appetizer": 1,
    "main": 8,
    "dessert": 14
}
```

Body · Cookies · Headers (5) · Test Results — 201 CREATED 9 ms 171 B

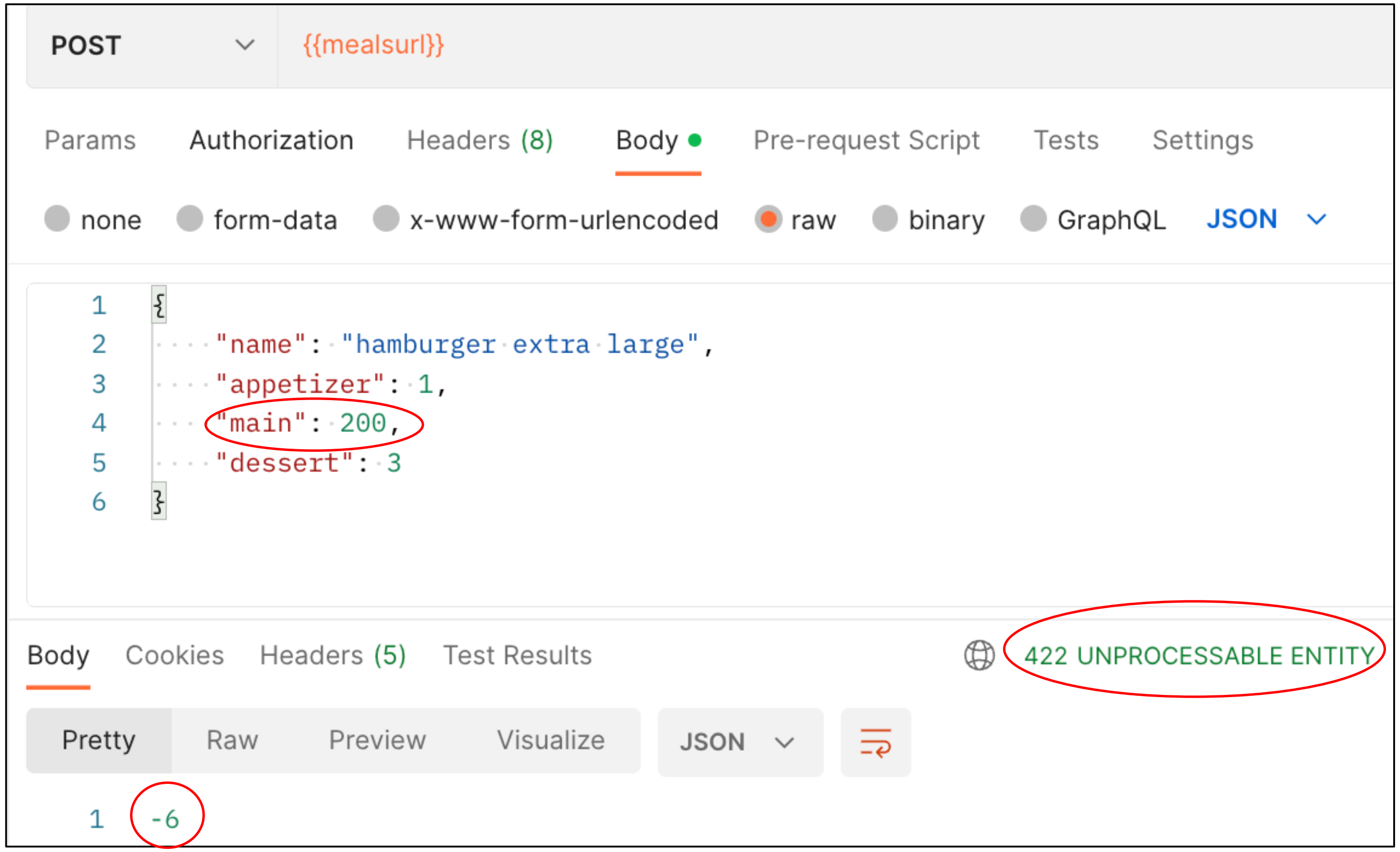
Pretty · Raw · Preview · Visualize · JSON

```
1
```

```
url = "http://0.0.0.0:80/meals"

payload = json.dumps({
  "name": "vegetarian",
  "appetizer": 1,
  "main": 8,
  "dessert": 14})

headers = {
  'Content-Type': 'application/json'}

response = requests.request("POST",
  url, headers=headers, data=payload)
```

Meal successfully created (response code 201). Returned ID for meal (in this case ID =1).

POST /meals with incorrect ID



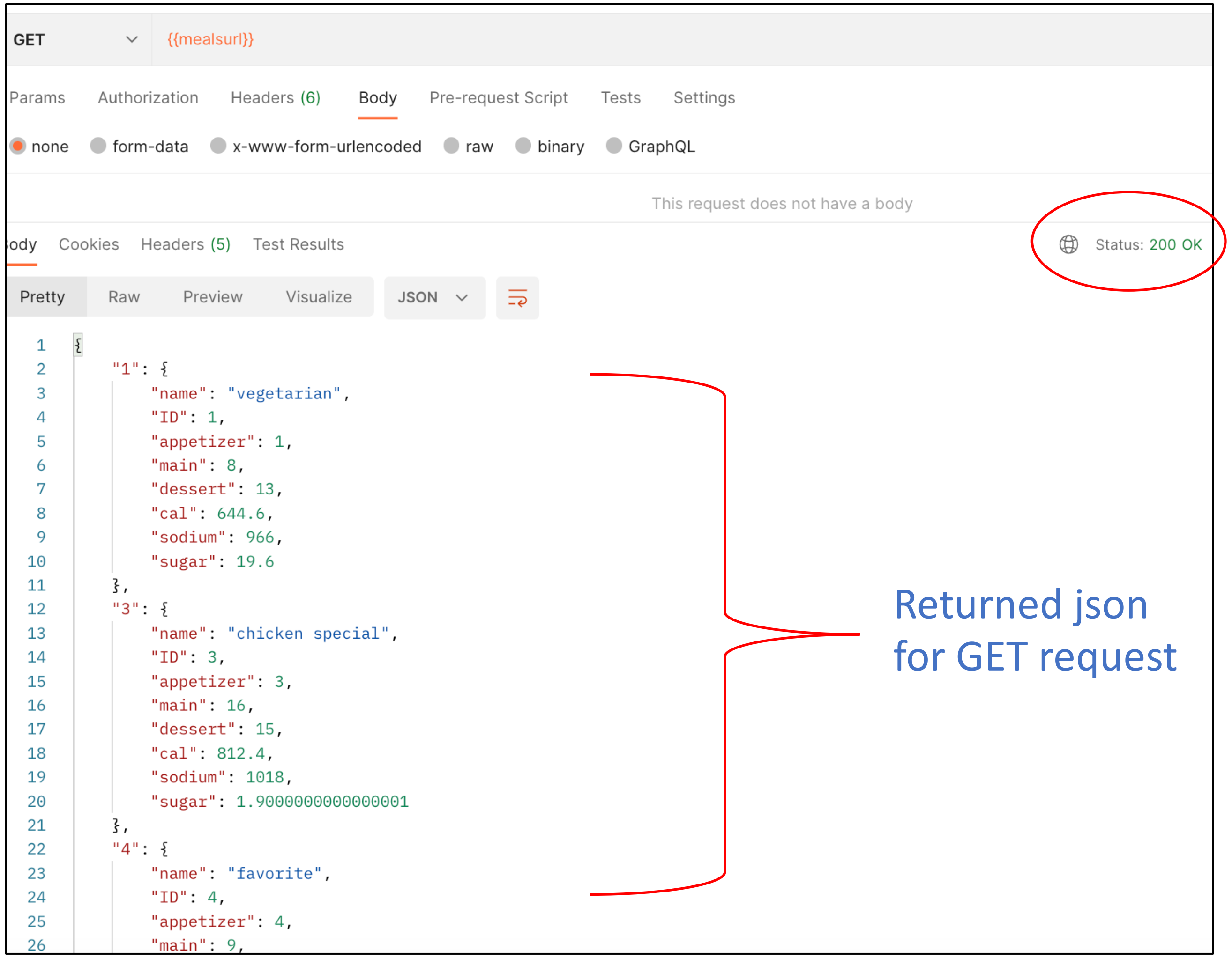
```
{
    "name": "hamburger extra large",
    "appetizer": 1,
    "main": 200,
    "dessert": 3
}
```

422 UNPROCESSABLE ENTITY

```
-6
```

JSON request where ID for “main” is not valid. Return ID = -5 and status code 422.

Get /meals



This request does not have a body

Status: 200 OK

```
{
    "1": {
        "name": "vegetarian",
        "ID": 1,
        "appetizer": 1,
        "main": 8,
        "dessert": 13,
        "cal": 644.6,
        "sodium": 966,
        "sugar": 19.6
    },
    "3": {
        "name": "chicken special",
        "ID": 3,
        "appetizer": 3,
        "main": 16,
        "dessert": 15,
        "cal": 812.4,
        "sodium": 1018,
        "sugar": 1.9000000000000001
    },
    "4": {
        "name": "favorite",
        "ID": 4,
        "appetizer": 4,
        "main": 9,
```

Returned json for GET request

```python
import requests

url = "http://0.0.0.0:80/meals"

payload={}
headers = {}

response = requests.request("GET", url,
  headers=headers, data=payload)

print(response.text)
```

Python request

Requests on the /meals/meal resource

There are two ways to specify a meal – its ID or its name

/meals/{ID}

- **GET** sends the ID of the meal and will receives back the JSON meal object.
- **DELETE** sends the ID of the meal and removes that meal from /meals. It returns the ID of the deleted meal.
- **PUT** acts exactly like **POST** except the ID of the meal is supplied. *NOTE: A successful PUT request returns response code 200, not 201.*

/meals/{name}

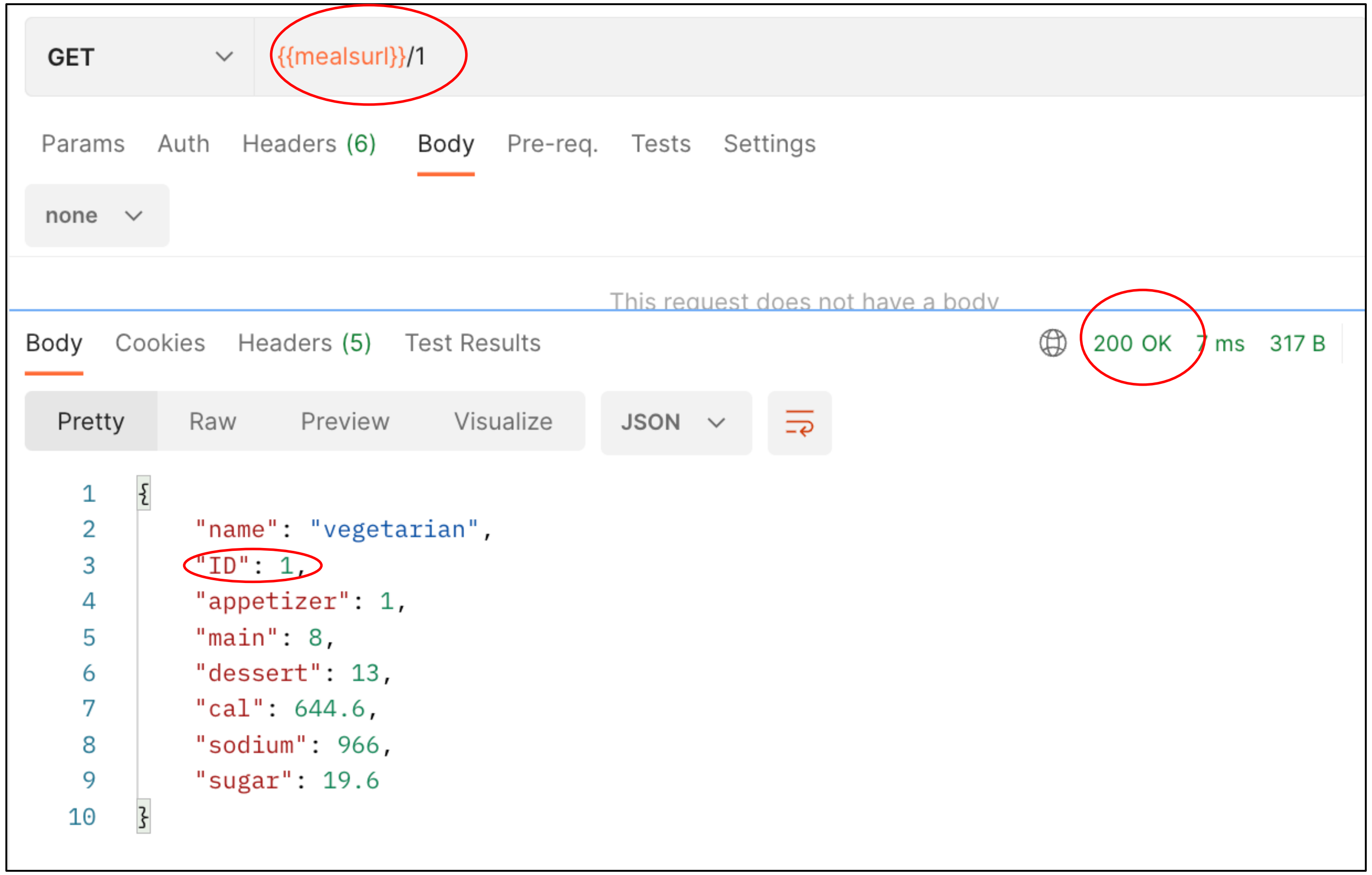
- **GET** and **DELETE** are the same as the above except instead of sending the ID of the meal, one sends the name of the meal.

GET and DELETE error responses

- If meal name or meal ID does not exist, the GET or DELETE request returns the response -5 with status code 404 (Not Found)

GET /meals/{ID}

```
curl --location 'http://0.0.0.0:80/meals/1'
```

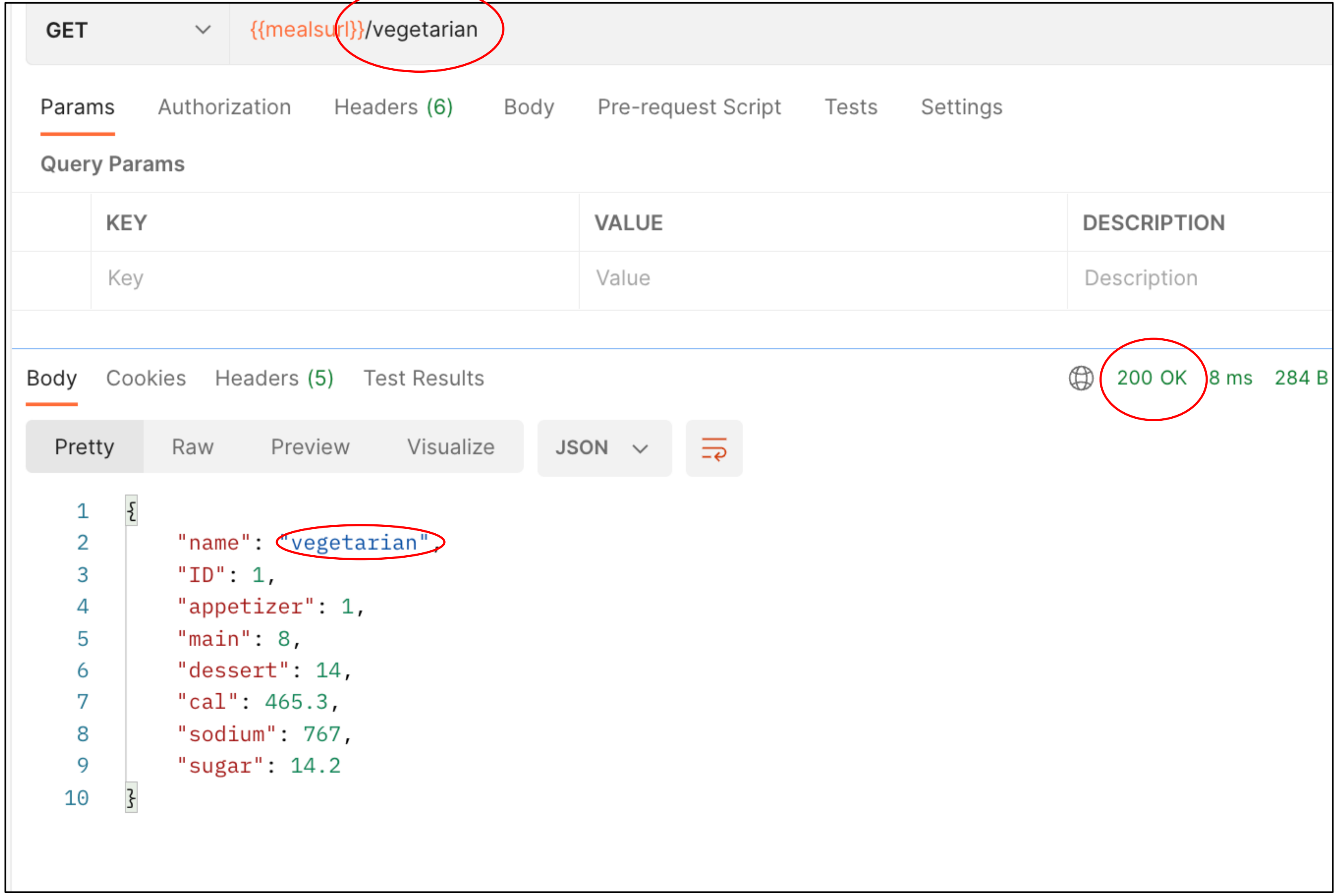


```
{
    "name": "vegetarian",
    "ID": 1,
    "appetizer": 1,
    "main": 8,
    "dessert": 13,
    "cal": 644.6,
    "sodium": 966,
    "sugar": 19.6
}
```

Returns the JSON object for meal with ID =1 and status code 200.

GET /meals/{name}

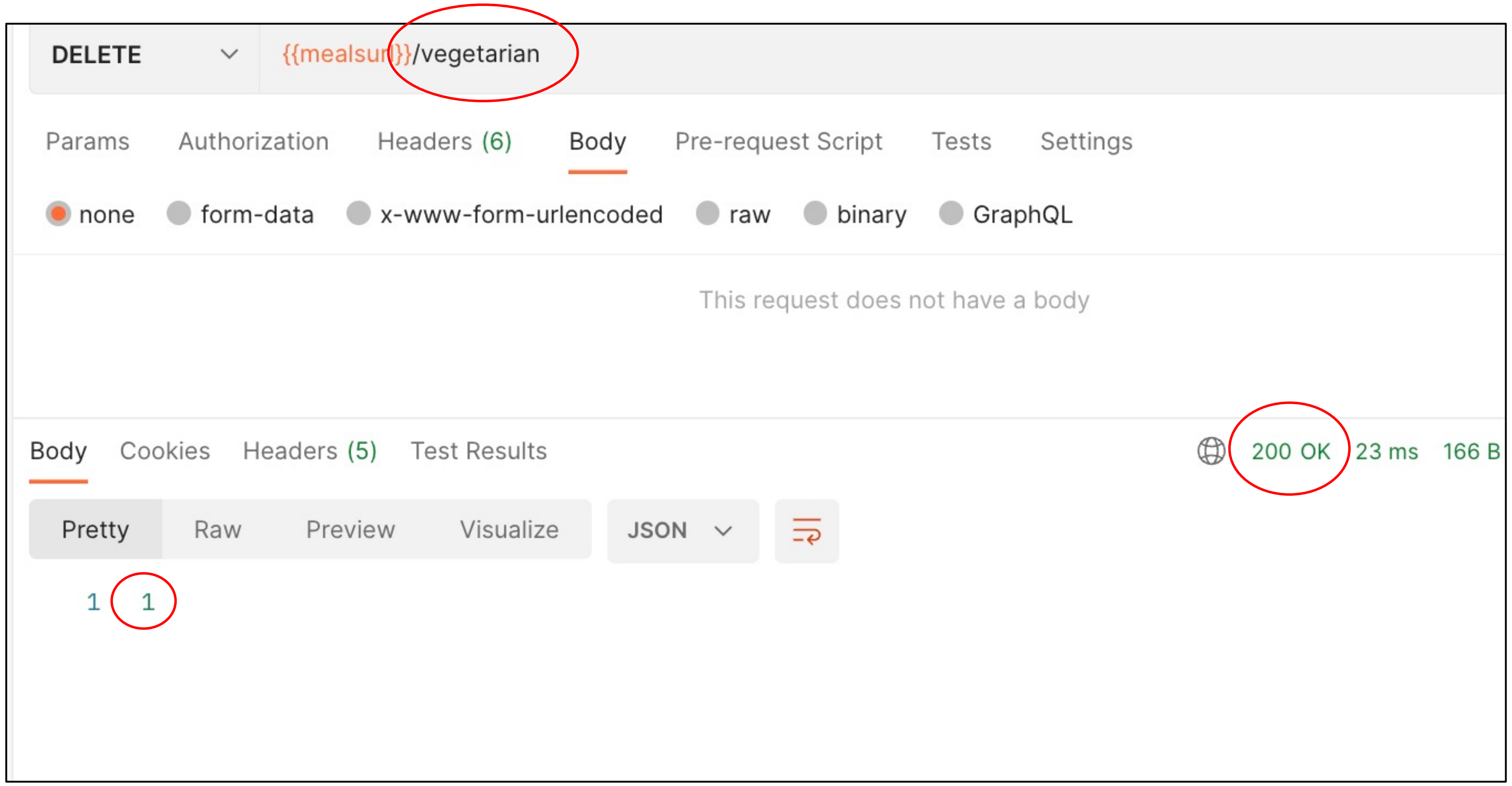
```
curl --location 'http://0.0.0.0:80/meals/vegetarian'
```



```
{
    "name": "vegetarian",
    "ID": 1,
    "appetizer": 1,
    "main": 8,
    "dessert": 14,
    "cal": 465.3,
    "sodium": 767,
    "sugar": 14.2
}
```

Returns the JSON object for meal with name = “vegetarian” and status code 200.

Delete /meals/{name}


DELETE {{mealsurl}}/vegetarian

● none ○ form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

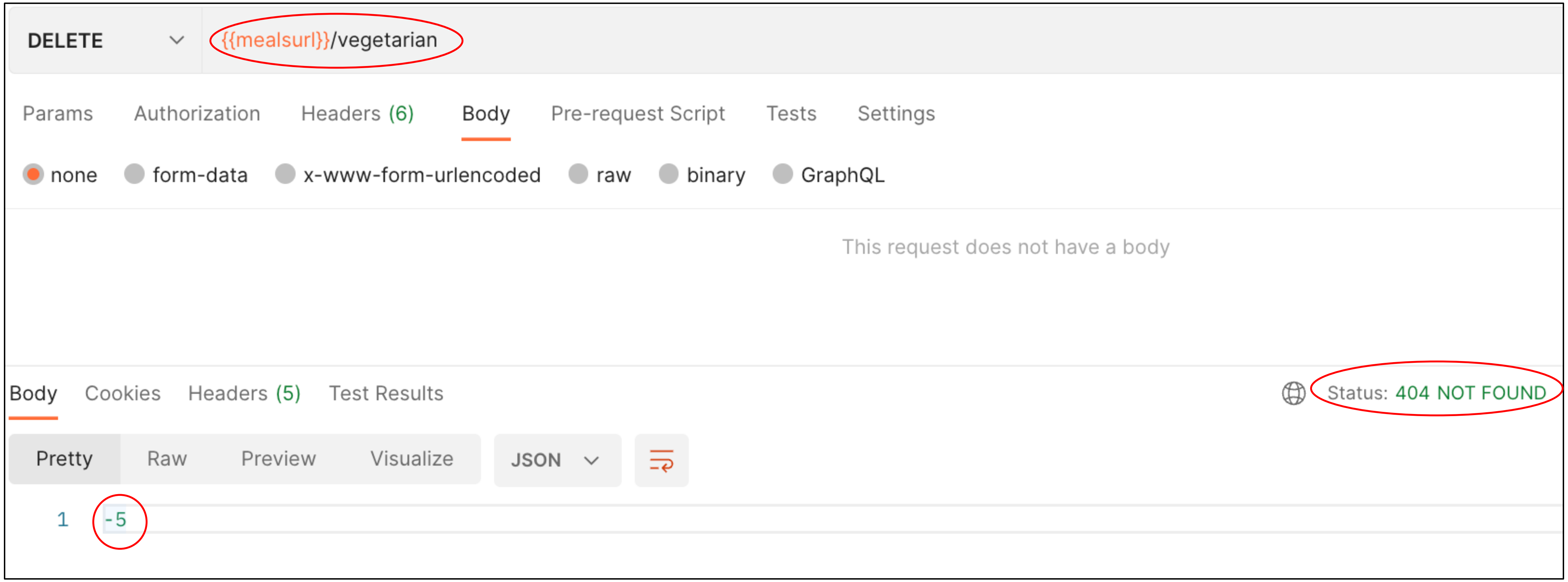
This request does not have a body

200 OK 23 ms 166 B

```
1
```

Deletes the meal with name = “vegetarian”. Returns the ID for that meal and status code 200.

DELETE /meals/{name} with incorrect name



DELETE {{mealsurl}}/vegetarian

Params Authorization Headers (6) Body Pre-request Script Tests Settings

[✓ none] form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (5) Test Results Status: 404 NOT FOUND

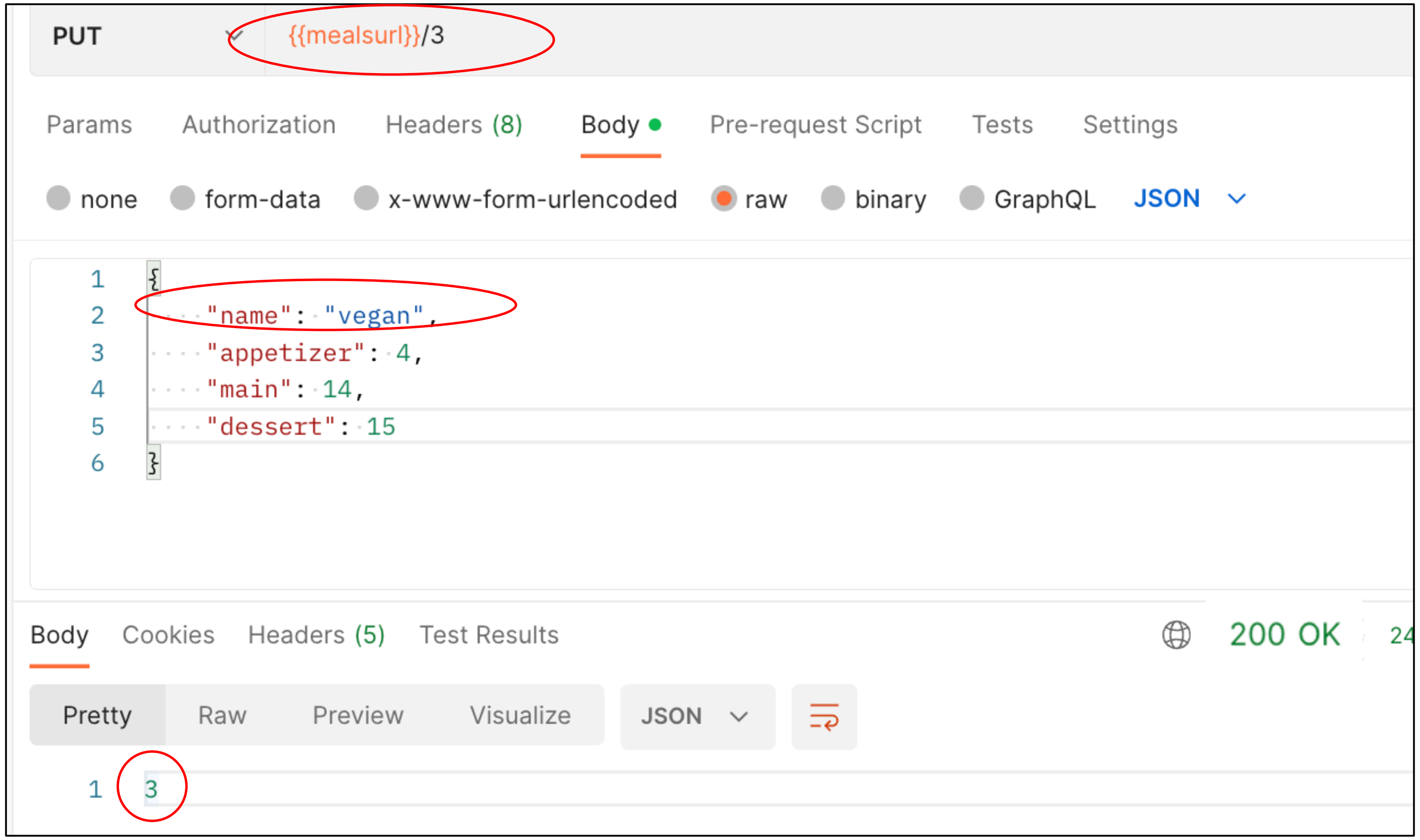
Pretty Raw Preview Visualize JSON

```
-5
```

The meal with name "vegetarian" was already deleted so returns error code -5 with code 404.

PUT /meals/{ID}

In this request, the meal with ID =3 (previously was named “chicken special”, see slide 30) is updated to have the name “vegan” and the appetizer and main courses are changed.



PUT {{mealsurl}}/3

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "name": "vegan",
    "appetizer": 4,
    "main": 14,
    "dessert": 15
}
```

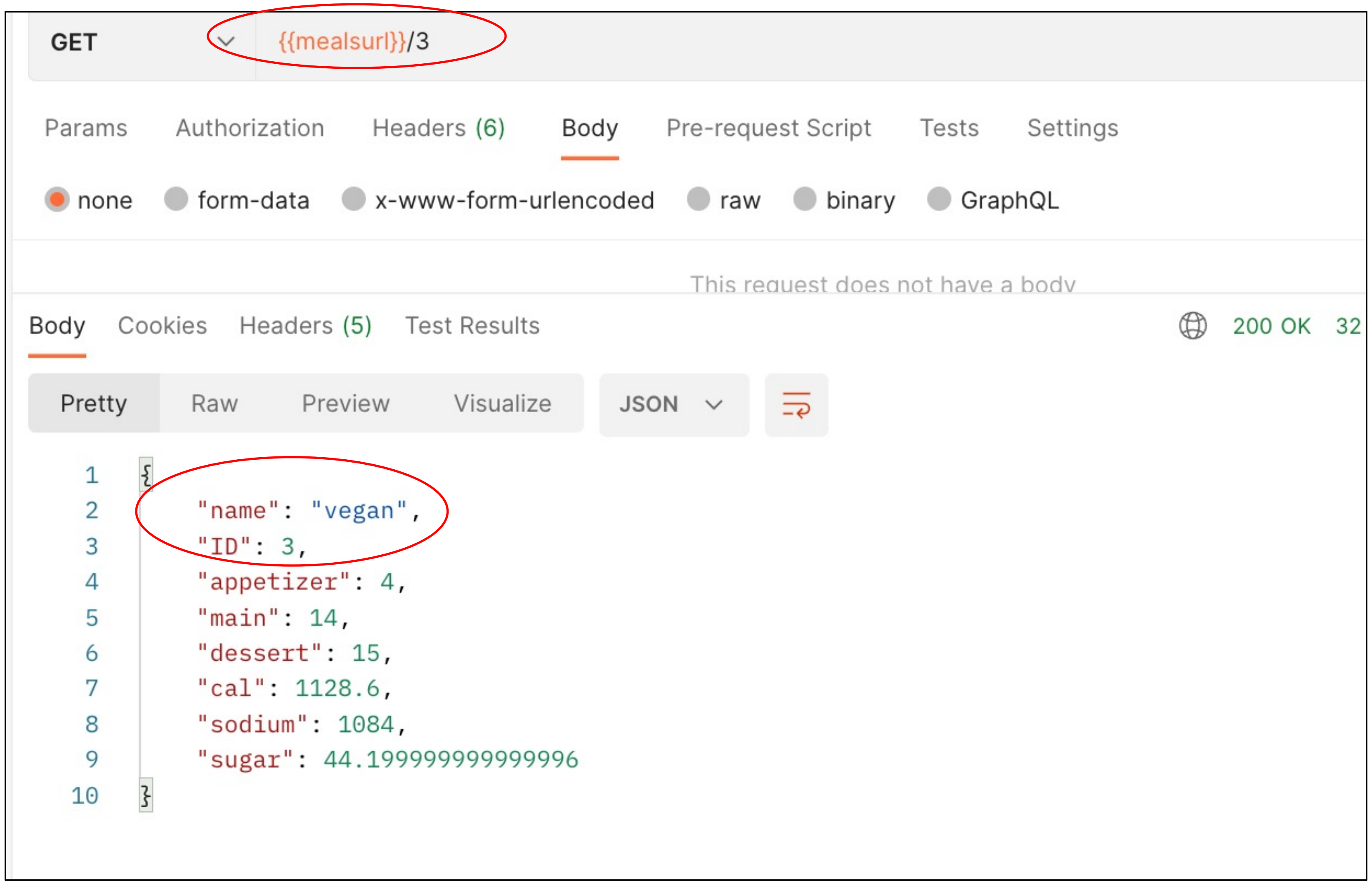
Body Cookies Headers (5) Test Results 200 OK 24

Pretty Raw Preview Visualize JSON

```
3
```

Note you only have to support PUT /meals/{ID} and not PUT /meals/{name}

After PUT, we check that it was successfully updated using request "GET /meals/3"



```
{
    "name": "vegan",
    "ID": 3,
    "appetizer": 4,
    "main": 14,
    "dessert": 15,
    "cal": 1128.6,
    "sodium": 1084,
    "sugar": 44.199999999999996
}
```

PUT to create a new meal

The slides entitled "More on POST, PUT, PATCH" given is on Topic-2 state that:

- A PUT request *creates* or *replaces* the resource given by the client specified URI
- In a PATCH request the client specifies the URI to updated. But it cannot create a new resources – the resources defined by the URI must *already exist* on the server.
- In this assignment, you can assume that **PUT will only be used for updating an existing meal,** not creating a new meal.

The effects of deleting a dish on a meal

What happens to a meal when one of its dishes is deleted?

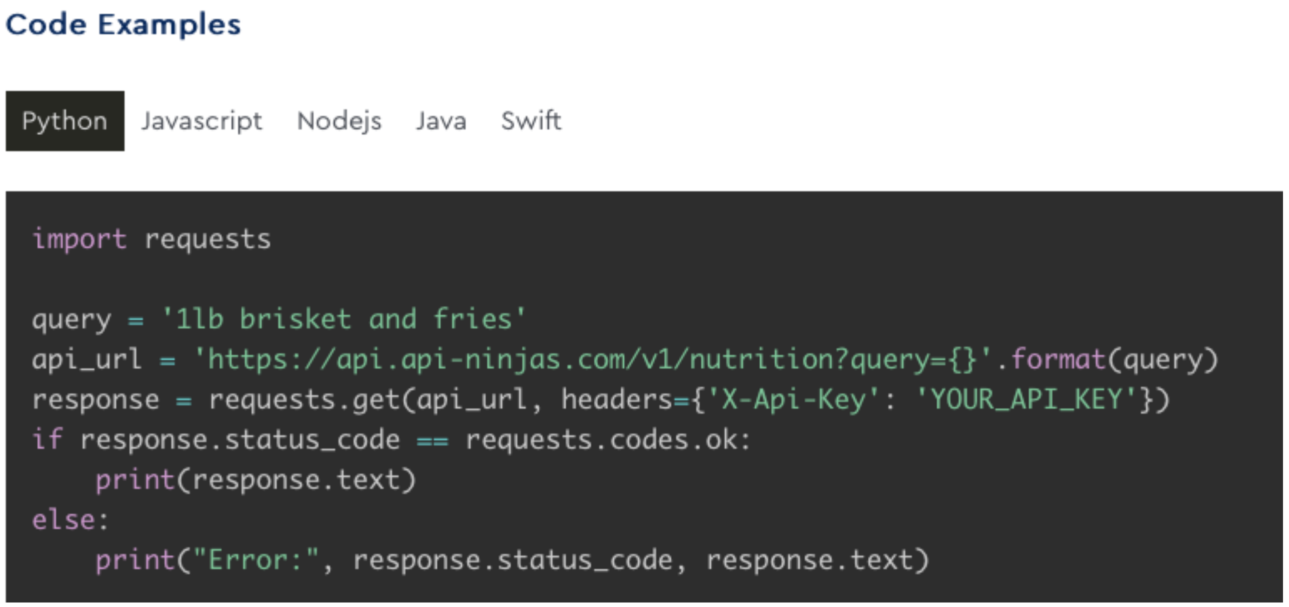
- If a dish is deleted, and that dish is part of a meal (appetizer, main or dessert) then that field of the meal should be set to the JSON **null** value (see https://www.json.org/json-en.html)
    - Note that this **null**, not “null”

- This does create an asymmetry.   You cannot create a meal with one course missing, but if you successfully create a meal, and then delete a dish, you can have a meal with one course missing.   Don’t worry about that for now.

Use api-ninjas to compute dish nutritional information

When the user adds a dish, the program will compute the *number of calories*, *the serving size*, and a *recipe* for that dish

- Use the API given at https://api-ninjas.com/api/nutrition
  - Interactive UI, documentation, and code snippets as well
- Given a short dish description, it will return nutritional information including the number of calories, sodium and sugar content, and the serving size (in grams)



```python
import requests

query = '1lb brisket and fries'
api_url = 'https://api.api-ninjas.com/v1/nutrition?query={}'.format(query)
response = requests.get(api_url, headers={'X-Api-Key': 'YOUR_API_KEY'})
if response.status_code == requests.codes.ok:
    print(response.text)
else:
    print("Error:", response.status_code, response.text)
```

Example code to use this API

https://api-ninjas.com/api/nutrition

- 50,000 API calls per month are free – that is far more than enough for this project.   https://api-ninjas.com/pricing
- YouTube on using API-Ninjas (I did not watch this videos so cannot attest to its quality.   There are other videos available – search for them if interested).
  - https://www.youtube.com/watch?v=QPTVTNqupr0
- You API-Ninjas API key must be included with your code or accessible from your code, as we will test your app using your key.
  - Make sure there are enough API calls remaining in your account for us to test with (say a few hundred).

Multiple responses from the nutrition API

- If the dish sent to https://api-ninjas.com/api/nutrition returns multiple components, then you should return the combined contents of the components.
- For example, “brisket and fries” returns a JSON array with two elements, one for “brisket” and one for “fries”. In this case, you would add the values, such as calories, for both “brisket” and “fries”. (The same is true for “sodium” and “sugar” and “serving size”).

Docker

The code you write will need to run in a Docker container.   This means that you need to submit:

  - A Dockerfile which automatically builds a docker image to run your program
  - Any files that the Dockerfile needs to build the image (which includes any code or auxiliary files you use)

- You should submit all of you code in a zip file.   If your Dockerfile assumes a specific file structure (e.g., it assumes that code on the host is in a subdirectory) then unzipping the zip file should preserve that directory structure.
- You should test that this works before submitting.   For instance, give your zip file to a friend to unzip and run the Docker build and run commands, and make sure that it works for your friend before submitting.

Docker

We will test your code by:

- Running your Dockerfile to build the image
- Creating a container from the image
- Issuing REST requests against the container on the specified port (next slide) and checking that the correct responses are returned.

Docker port

- You should have your API server application listen for HTTP (API) requests on port 8000
  - In otherwords, the Docker container port is 8000. When we test the program, we will forward requests (using --publish) from the host to the container port 8000.
  - It should be possible to change the port the application (container) is listening on very simply.
- You should submit your source code as part of the zip file, even if you do not need it to run your application (e.g., you are using compiled code for instance).
  - The names of the submitters should be the name of the ZIP file
    E.g., Mayan_Shem_Hen_Tavi