Assignment #2
HA and robust applications

Dr. Daniel Yellin

Assignment #2

In this assignment you will have to:

1. Use **Docker Compose** to create an application that is built from 4 services:
   - the meals service you did for assignment #1 and a *diet* service you will build,
   - a database service and a reverse-proxy service that you download from DockerHub
2. Make the meals and the diet services **persistent**
3. Use Docker Compose to **restart the assignment and diet services after a failure** (and process requests as if it never failed)
4. Use a **reverse-proxy** (NGINX) to route requests to the right server
5. **Extra-credit**: implement **load balancing** for the meal service

The diet microservice

The diet service supports:

- POST /diets
- GET /diets
- GET /diet/name

A **diet** is a JSON object of the form:

```
{
      ”name”: <string>,
      “cal” : <number>,
      “sodium” : <number>,
       “sugar” : <number>
}
```

Example:


```
{
"name": "low sodium",
"cal": 5000,
"sodium": 5,
"sugar": 50
}
```


NOTE: **No ID** is included in the JSON object

Modify the GET /meals request of the meals service

Modify the meals service to support:

GET /meals
‘http://0.0.0.0:port/meals?diet=<name>’

where <name> gives the name of a specific diet. The response are all those meals that conform to the diets.

If the diet specifies

cal=num1,sodium=num2,&
sugar=num3

then all the meals returned have calories <= num1, sodium <= num2, and sugar <=num3

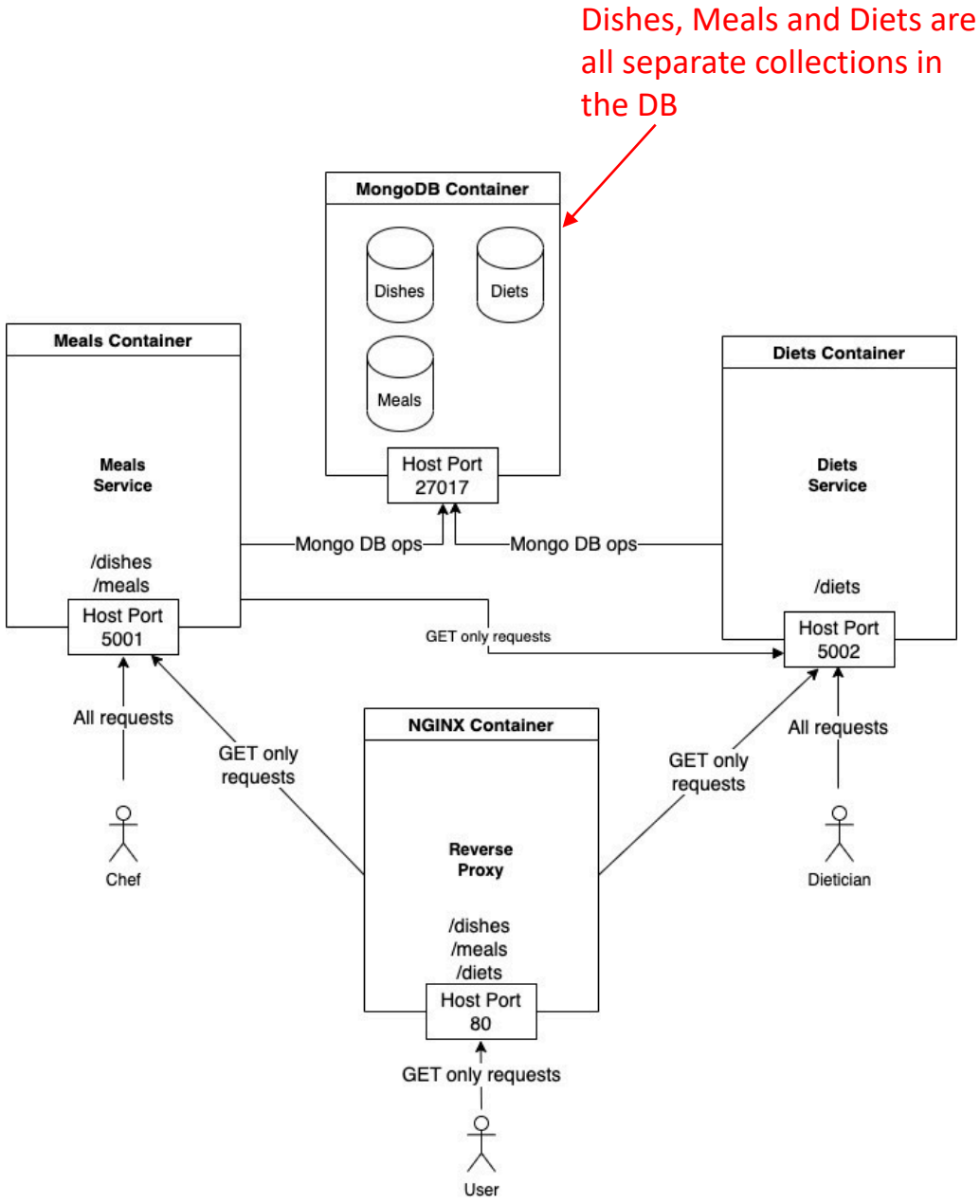
Note: GET /meals is still supported with no query string

Clarification

- When assigning unique IDs to dishes, meals, and diets, you **cannot reuse** an ID.
  - E.g., if a meal M was assigned the ID X, then M was deleted, you cannot reuse the ID X for another meal that gets added later.

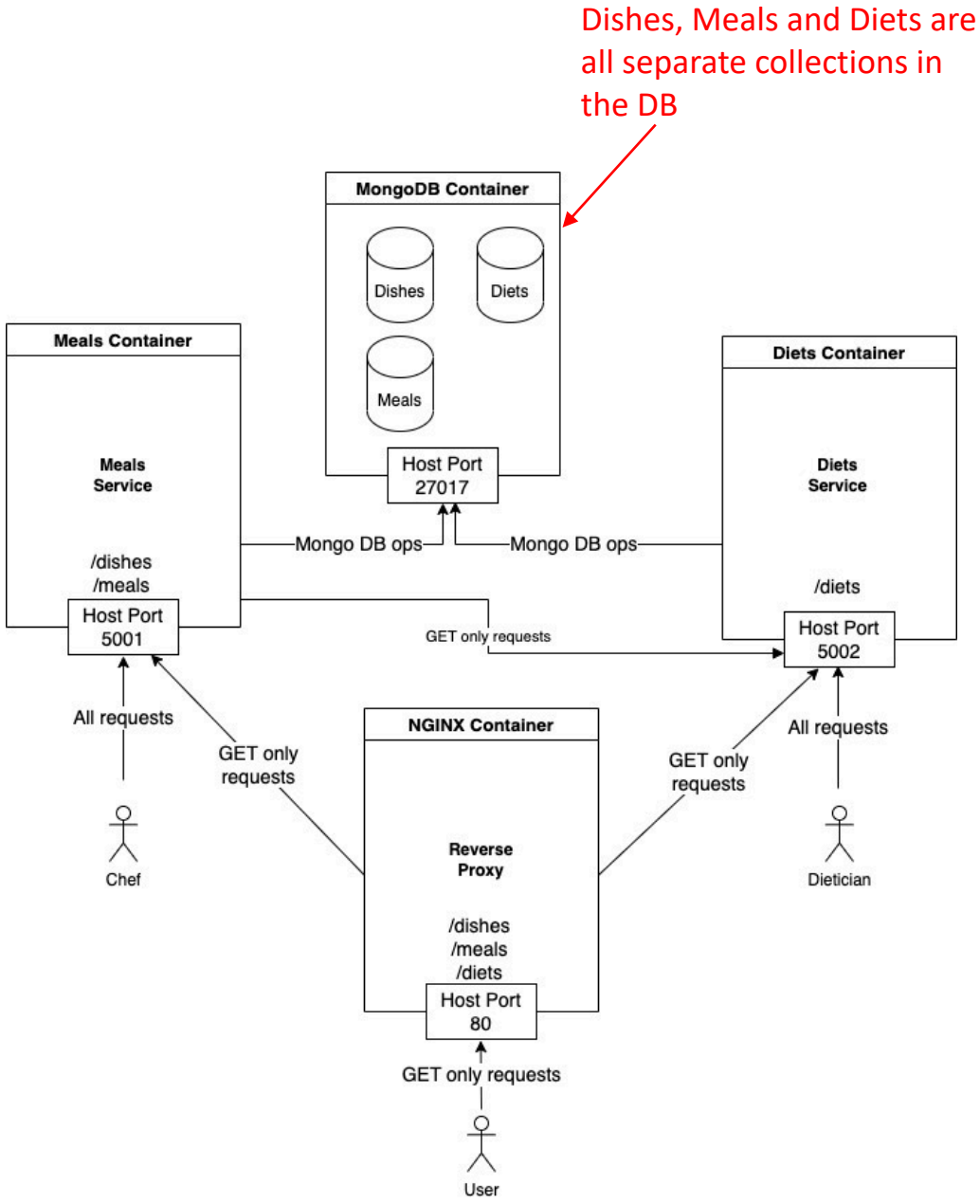
Architecture

- Create a docker-compose.yml that implements this architecture
  - It must include instructions for restarting services when they fail
  - It must have the services responding to requests on the host ports listed
  - It must start the services in the appropriate order
- Implement persistence for the meals and diet microservices



Architecture (cont)

- The meals and diets servics must restart in the appropriate state after a failure:
  - Requests after a failure should be answered as if there was no failure
- Implement the reverse proxy as in the diagram
  - Requests on port 80 for meals or diets only support GET requests
  - Requests on port 5001 (5002) that do not go through the reverse-proxy support all requests (GET,POST,PUT…)



How to invoke a container API from another container

Assume we have two services, A-svc and B-svc.

- A-svc provides a REST API for a resource “/my_resource”.
- The A-svc declares the port mapping “4017:8090”.
- Using the docker-compose.yml given here, how would the B-svc invoke this REST API; e.g., with a GET request?

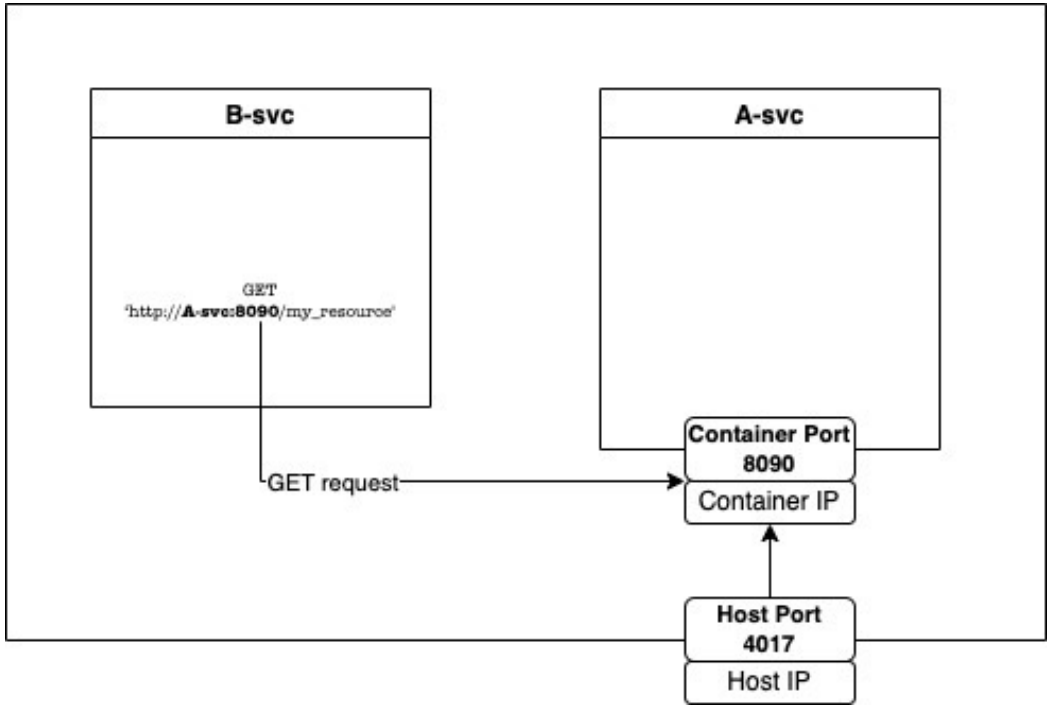
```
version: '3'  # version of compose format

services:
  A-svc:
    build: ./app
    …
    ports:
       - "4017:8090"
    expose:

       - 8090

B-svc:
    build: ./dir

    …
```

How to invoke a container API from another container (cont)

For the B-svc to invoke the API provided by the A-svc, it would issue the cmd:

GET 'http://**A-svc:8090**/my_resource'

- Since containers are isolated from the host, it needs to invoke the API from inside Docker, using the Docker network.
- **"A-svc"** is a symbolic name that refers to the Docker IP of the container running the A-svc service.
- Since this invocation is not going through the host, but directly to the container's IP inside Docker, the port must be the container port 8090.



How to invoke a container API from another container (cont)

IMPORTANT: you must use the expose command on the container IP (8090) to allow other container services to reach this port.

```
version: '3'  # version of compose format

services:
  A-svc:
    build: ./app
    ...
    ports:
      - "4017:8090"
    expose:
      - 8090

B-svc:
    build: ./dir
    ...
```

The meals service

- The meals service supports all the same requests as in assignment #1. While it still supports GET /meals, without any query string, it also now supports
  - GET URI:port/meals?diet=<name>
    This request returns those meals that conforms to the diet.

One other change to dishes and meals API

GET /dishes and GET /meals will return a **JSON array** of JSON objects

- **NOT** a JSON object (where each field was also a JSON object).
- Note that the ID of the dish (meal) is part of the JSON object.
- Note, to make it easier to use with Mongo, instead of “ID” field we have “_id:” field.


```
[
    {
        "_id": 1,
        "name": "green salad",
        "cal": 23.4,
        "size": 100.0,
        "sodium": 37,
        "sugar": 2.2
    },
    {
        "_id": 2,
        "name": "scrambled eggs",
        "cal": 150.2,
        "size": 100.0,
        "sodium": 146,
        "sugar": 1.4
    },
    {
        "_id": 3,
        "name": "brisket",
        "cal": 289.3,
        "size": 100.0,
        "sodium": 47,
        "sugar": 0.0
    }
]
```

The diets service

The POST /diets request provides a diet JSON object. It returns the following:

- If request is successful, it returns “Diet <name> was created successfully” with a status code of 201
- If the <name> provided for the diet is already in use (was already added), it returns “Diet with <name> already exists” with a status code of 422.
- If the JSON object is ill-formed such as not containing a “cal”, “sodium” or “sugar” field, then it returns “Incorrect POST format” with a status code of 422.
- If the POST request did not supply JSON content, then it returns “POST expects content type to be application/json” with a status code of 415.

The diets service

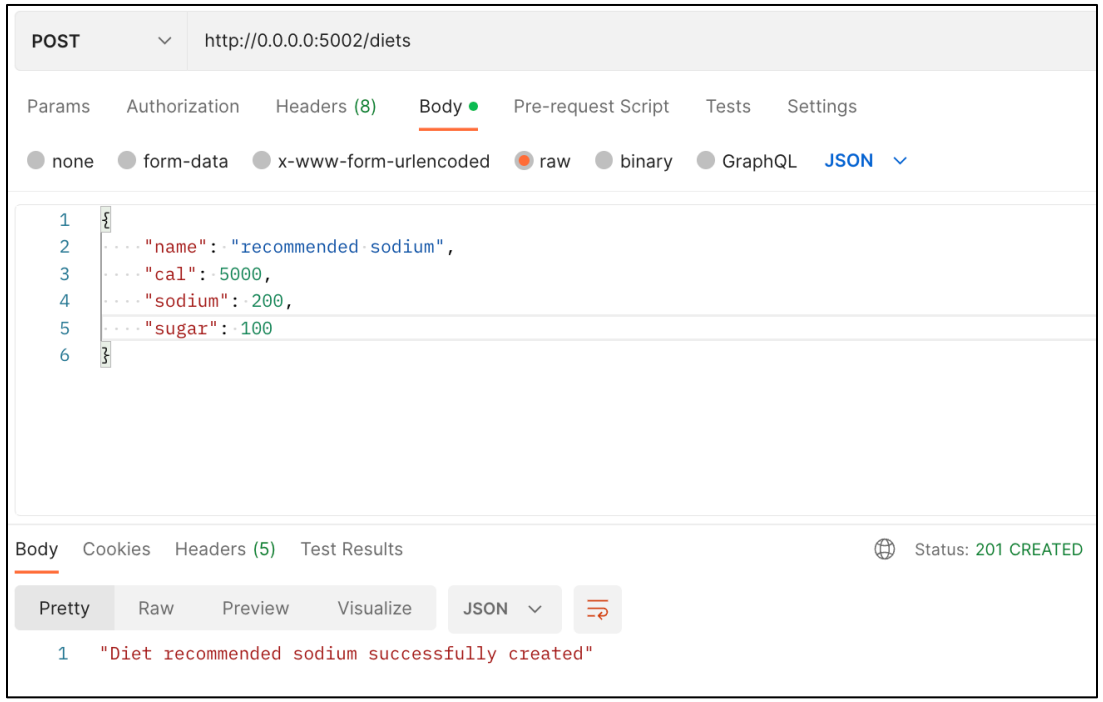
The POST /diets request provides a diet JSON object. It returns the following:

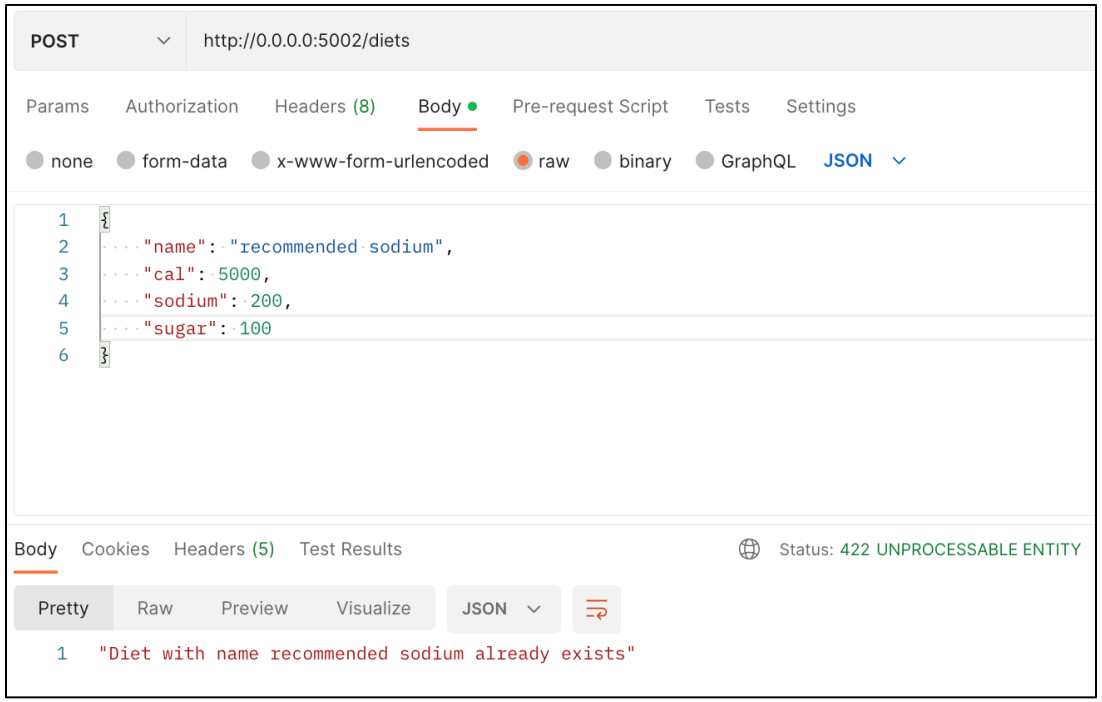
- If request is successful, it returns “Diet <name> was created successfully” with a status code of 201
- If the <name> provided for the diet is already in use (was already added), it returns “Diet with <name> already exists” with a status code of 422.

You are only responsible for these two sorts of requests:

1. Where Diet is successfully created, and
2. Where the name of the diet is already in use.

Successful and unsuccessful POST /diets requests

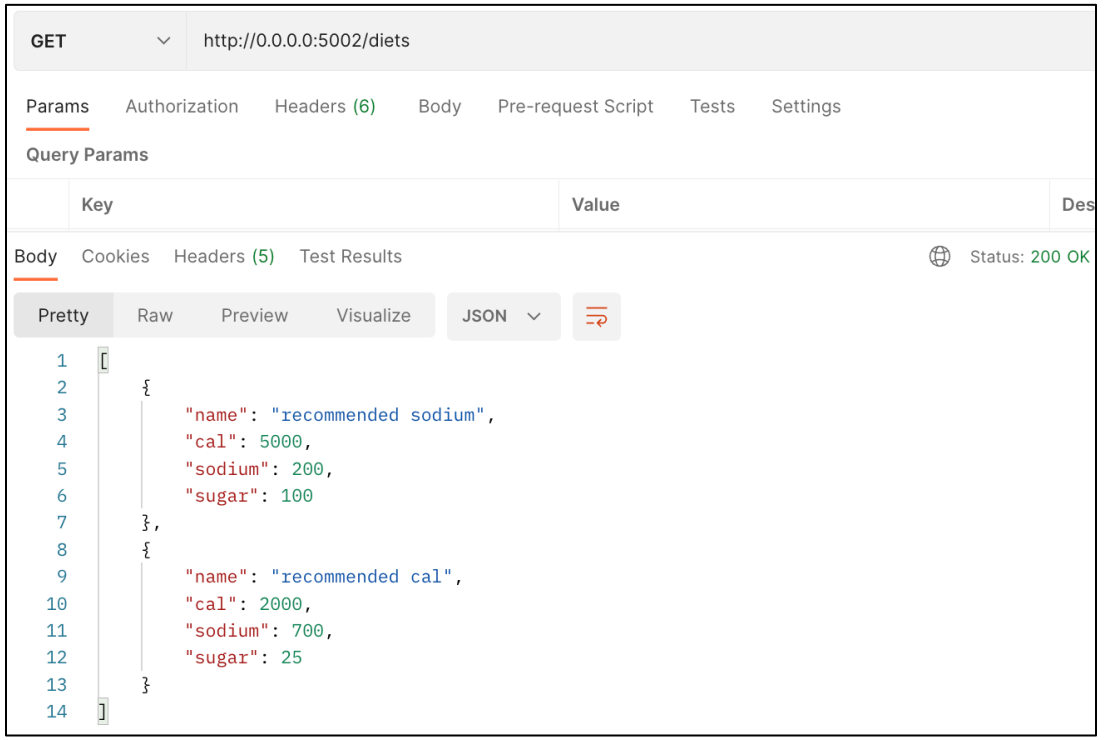




The diets service (cont)

The GET /diets request returns the following:

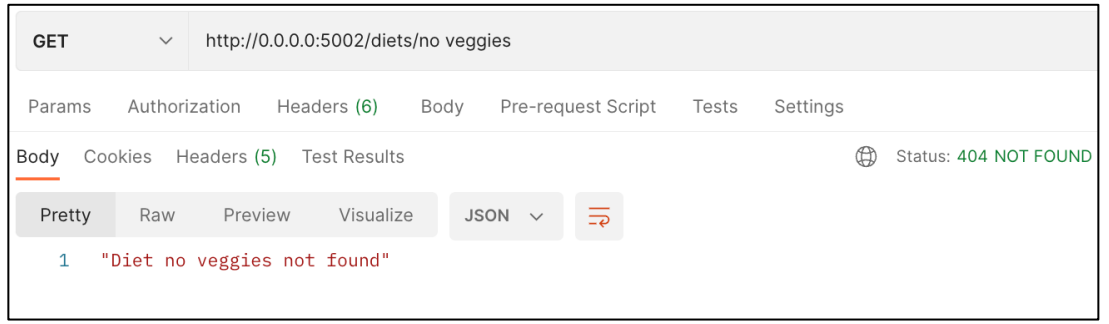
- It returns a JSON array of all diets with a status code 200
- NOTE: no “_id” field is returned. Eventhough Mongo creates such a field, for the end-user, this is not relevant and is not included in the output. (See slide 3).



The diets service (cont)

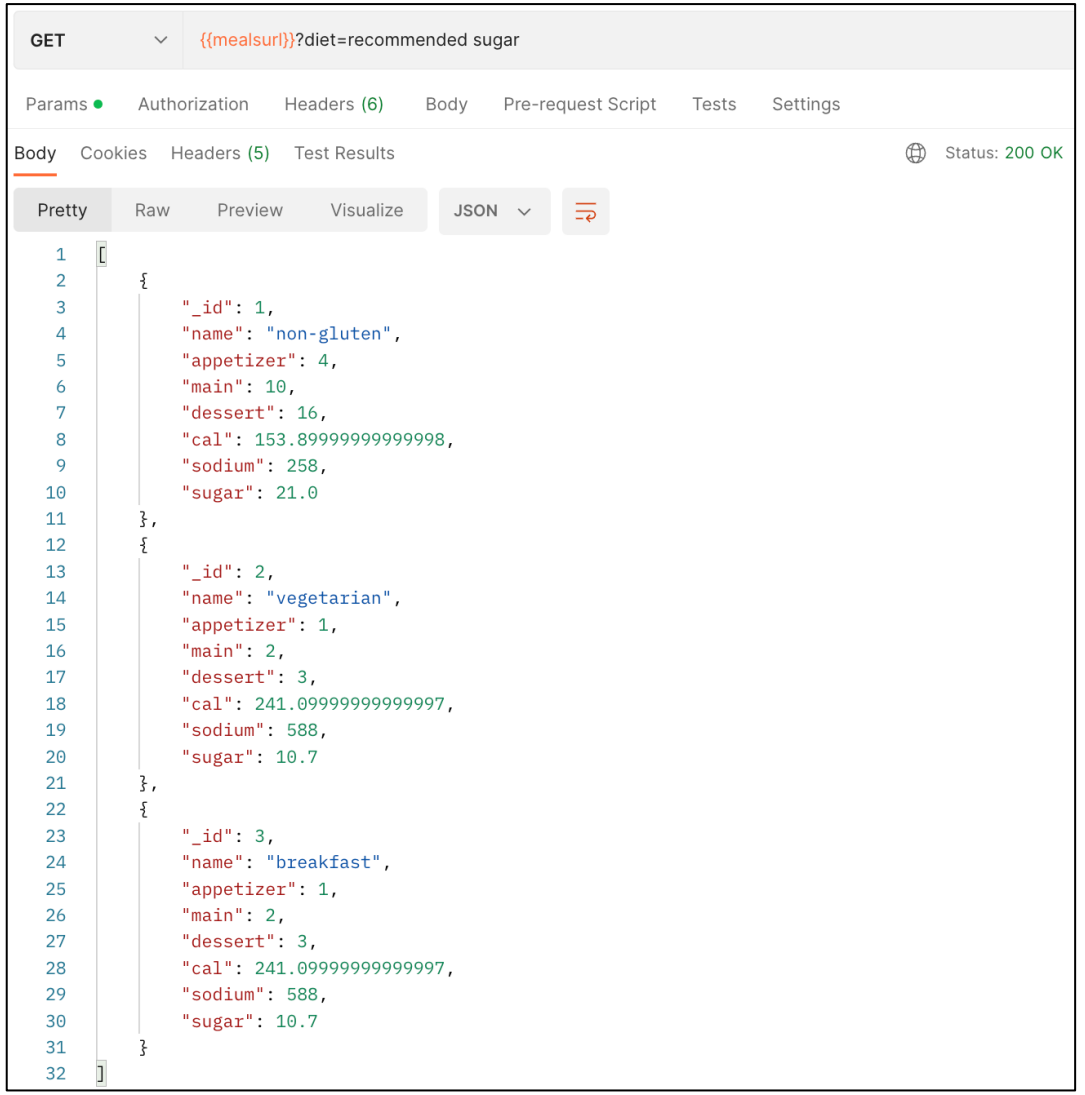
The GET /diets/{diet_name} request returns the following:

- If the request is successful, it returns the requested JSON object
- If there does not exist a diet of the given name, it returns “Diet <diet_name> not found” with status code 404 (example shown on right).



Meals with query string

This example shows invoking the /meals API returning only those meals that satisfy the diet specified in the query string.



Killing containers

- Make sure that in your container images for meals as well as diets, you have **/bin/sh** available. You probably do. To check, do the following:
  - Start up your container and issue the cmd
    docker exec --it <container> /bin/sh
    where <container> is the name or ID of the container
  - If after executing this command you are inside your container and can issue commands (try "**ls**" for example), then you are good.
- The בודק will use /bin/sh to kill your container and see if it restarts