

Assignment #2

HA and robust applications

Dr. Daniel Yellin

THESE SLIDES ARE THE PROPERTY OF DANIEL YELLIN
THEY ARE ONLY FOR USE BY STUDENTS OF THE CLASS
THERE IS NO PERMISSION TO DISTRIBUTE OR POST
THESE SLIDES TO OTHERS

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 /diets/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 \leq num1, sodium \leq num2, and sugar \leq num3

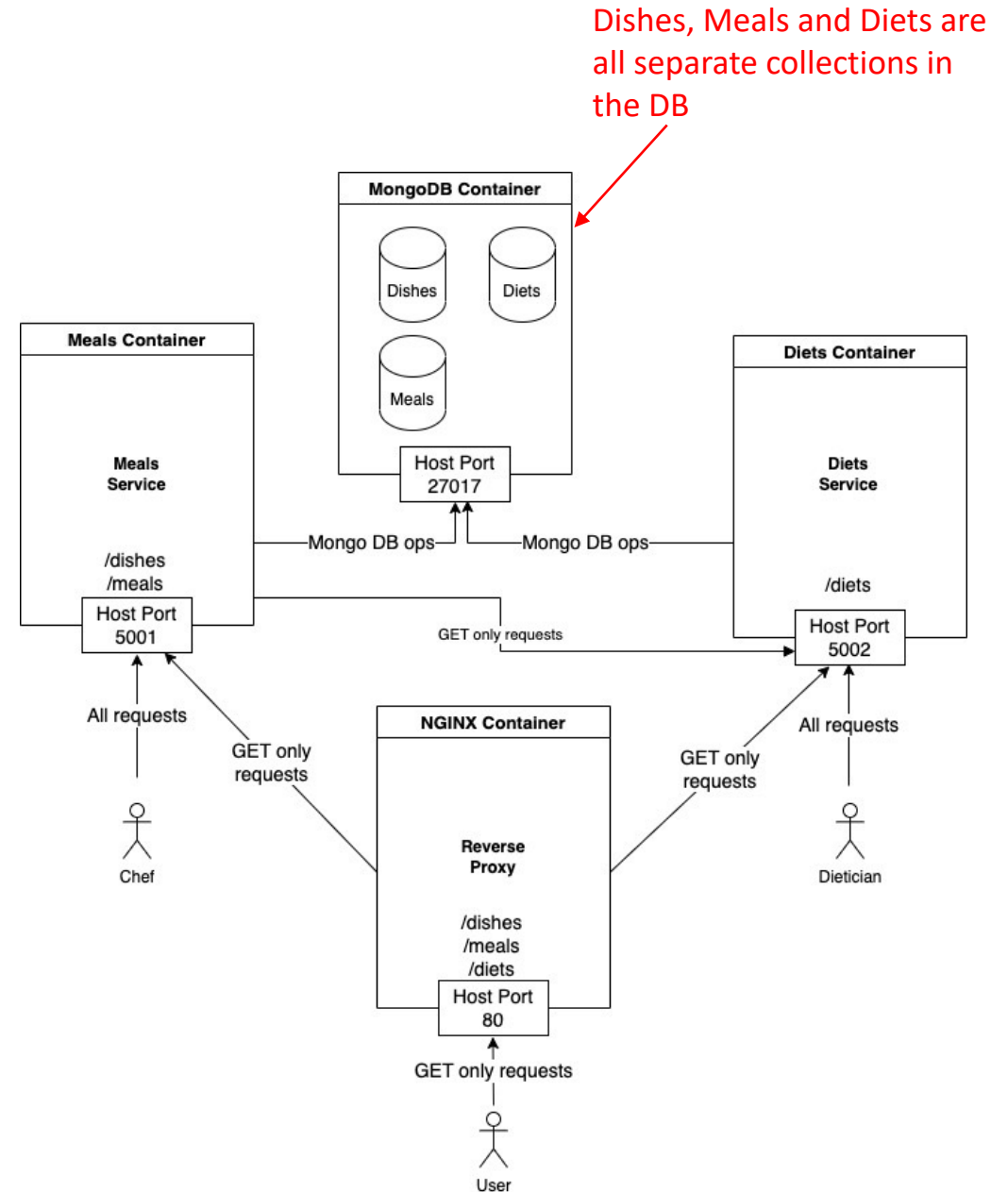
Note: GET /meals is still supported with no query string

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.

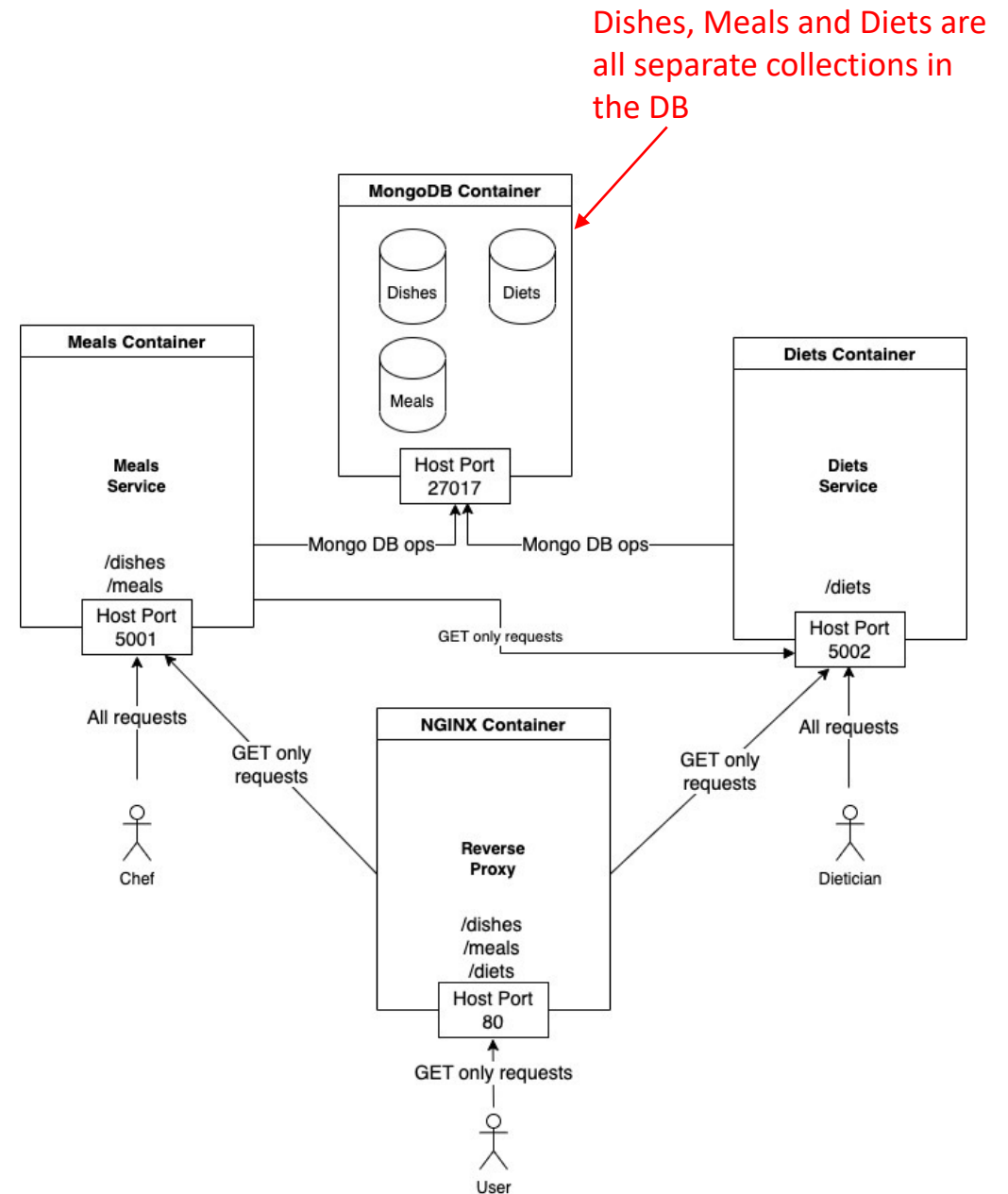
Architecture

- 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 services 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

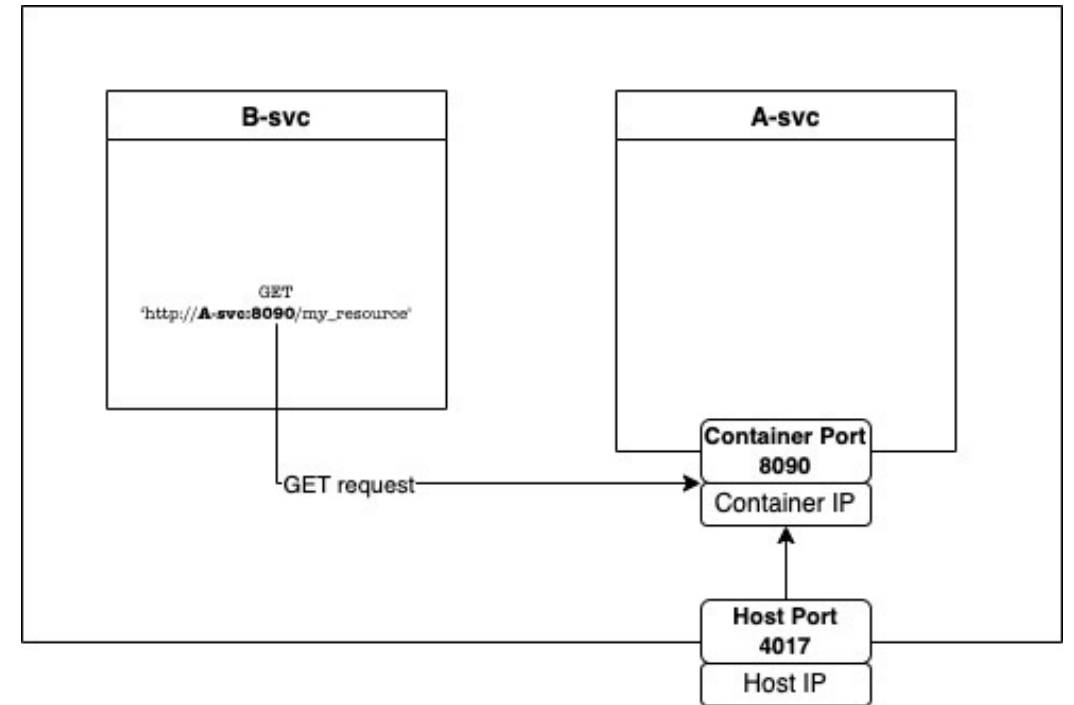
services:
  A-svc:
    build: ./app
    ...
    ports:
      - "4017:8090"
    expose:
      - 8090
  B-svc:
    build: ./dir
    ...
```


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.
 - If the <name> specified is not a known diet, then the service returns None with an error status code 422.

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.
- You DO need to return an integer ID (not the Mongo _id) for each dish and meal

```
[
  {
    "name": "waldorf salad",
    "cal": 174.7,
    "size": 100.0,
    "sodium": 128,
    "sugar": 8.7,
    "ID": 1
  },
  {
    "name": "garlic bread",
    "cal": 339.5,
    "size": 100.0,
    "sodium": 536,
    "sugar": 3.7,
    "ID": 2
  },
  {
    "name": "chicken soup",
    "cal": 33.2,
    "size": 100.0,
    "sodium": 230,
    "sugar": 1.0,
    "ID": 3
  }
]
```

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

POST ▼ http://0.0.0.0:5002/diets

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

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {
2   ... "name": "recommended-sodium",
3   ... "cal": 5000,
4   ... "sodium": 200,
5   ... "sugar": 100
6 }
```

Body Cookies Headers (5) Test Results 🌐 Status: 201 CREATED

Pretty Raw Preview Visualize **JSON** ▼ 🔗

```
1 "Diet recommended sodium successfully created"
```

POST ▼ http://0.0.0.0:5002/diets

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

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {
2   ... "name": "recommended-sodium",
3   ... "cal": 5000,
4   ... "sodium": 200,
5   ... "sugar": 100
6 }
```

Body Cookies Headers (5) Test Results 🌐 Status: 422 UNPROCESSABLE ENTITY

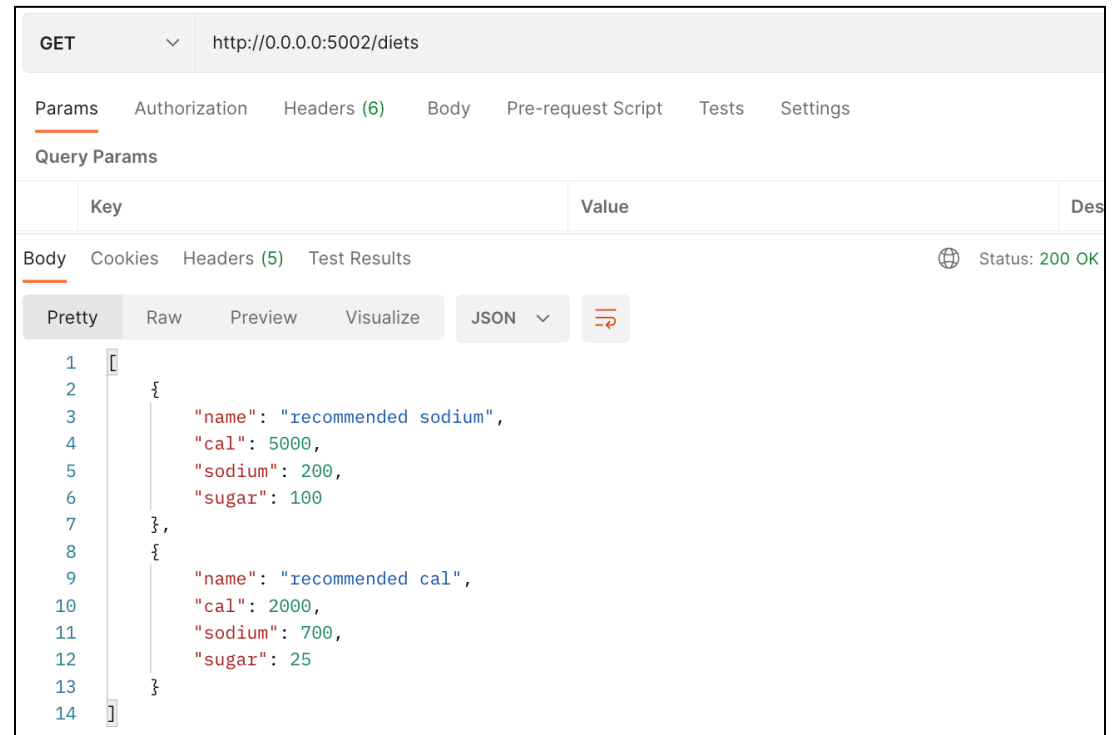
Pretty Raw Preview Visualize **JSON** ▼ 🔗

```
1 "Diet with name recommended sodium already exists"
```

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).



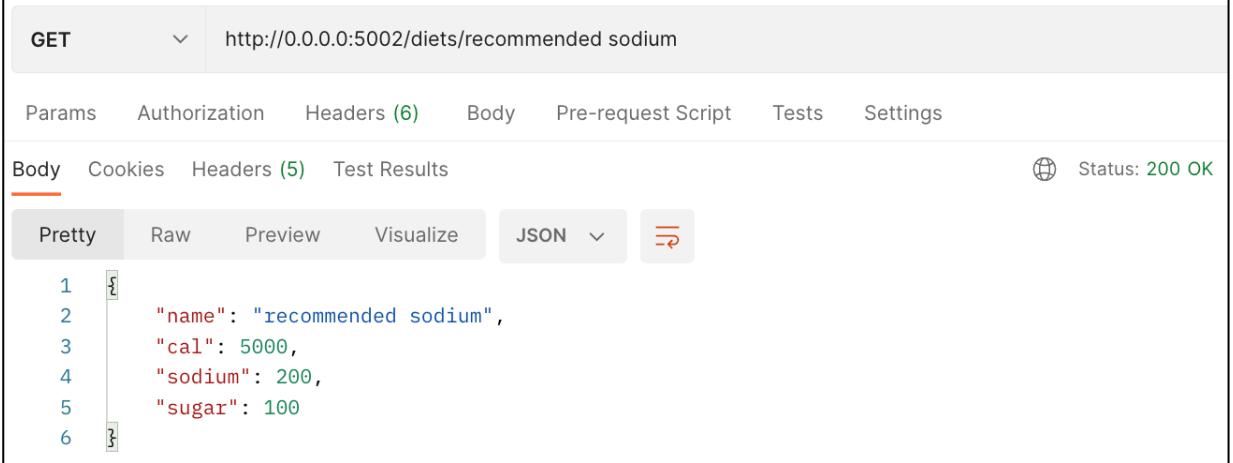
```
GET http://0.0.0.0:5002/diets

Params Authorization Headers (6) Body Pre-request Script Tests Settings
Query Params
Key Value Des
Body Cookies Headers (5) Test Results Status: 200 OK
Pretty Raw Preview Visualize JSON
1 {
2   {
3     "name": "recommended sodium",
4     "cal": 5000,
5     "sodium": 200,
6     "sugar": 100
7   },
8   {
9     "name": "recommended cal",
10    "cal": 2000,
11    "sodium": 700,
12    "sugar": 25
13  }
14 }
```

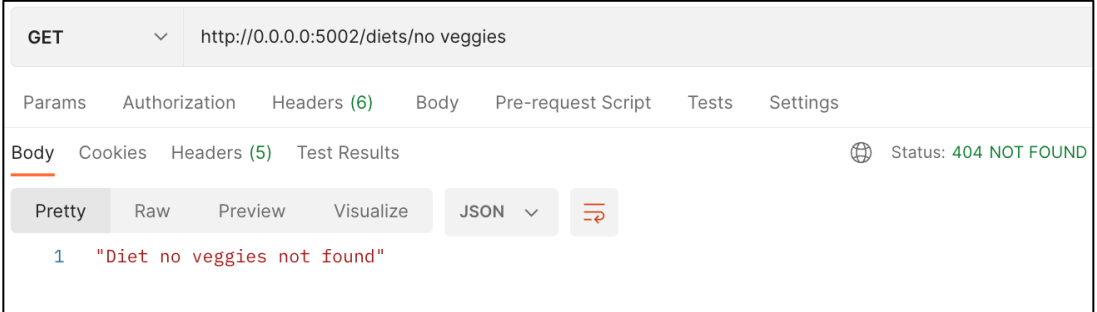

The diets service (cont)

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).



```
GET http://0.0.0.0:5002/diets/recommended sodium
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 {
2   "name": "recommended sodium",
3   "cal": 5000,
4   "sodium": 200,
5   "sugar": 100
6 }
```



```
GET http://0.0.0.0:5002/diets/no veggies
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Body Cookies Headers (5) Test Results Status: 404 NOT FOUND
Pretty Raw Preview Visualize JSON
1 "Diet no veggies not found"
```

Meals with query string

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

Note that the IDs of the appetizer, main and dessert are also integers.

The screenshot shows a REST client interface for a request named "assignment 2 / get two times sodium". The request method is GET, and the URL is `{{mealsurl}}?diet=two times sodium`. The "Query Params" tab is active, showing a single parameter: `diet=two times sodium`. The status of the request is 200 OK. The response body is displayed in the "Body" tab, showing a JSON array of two meal objects. The first object is for a "non-gluten" meal with ID 1, and the second is for an "alternative breakfast" meal with ID 5. Both meals have integer IDs for their appetizer, main, and dessert components.

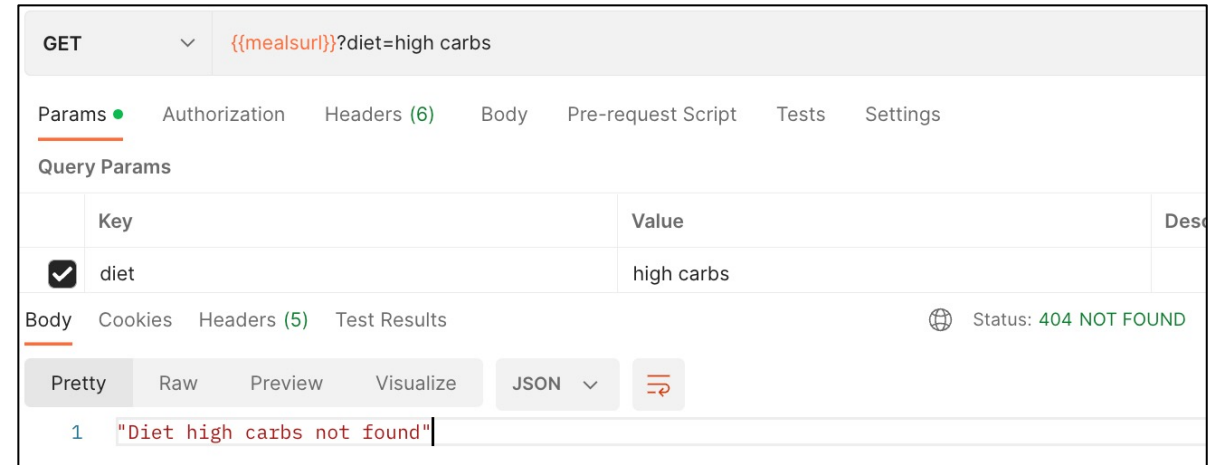
```
1 [
2   {
3     "name": "non-gluten",
4     "appetizer": 4,
5     "main": 10,
6     "dessert": 16,
7     "cal": 153.89999999999998,
8     "sodium": 258,
9     "sugar": 21.0,
10    "ID": 1
11  },
12  {
13    "name": "alternative breakfast",
14    "appetizer": 15,
15    "main": 17,
16    "dessert": 16,
17    "cal": 515.2,
18    "sodium": 273,
19    "sugar": 62.300000000000004,
20    "ID": 5
21  }
22 ]
```

Meals with query string (cont)

In the screenshot on the right, the API requests a diet in the query string that does not exist (i.e., the diet service does not contain this diet).

The meals service should respond (as it is informed by the diet service) that it a diet of that name does not exist with status code 404.*

*One can argue that this is not the appropriate status code, but for this assignment, we will use it.



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