Assignment #2
HA and robust applications

Dr. Daniel Yellin

Due date

The assignment is due December 18, 2025 before 23:59

יש להגיש את המטלה עד 18.12.2025
עד שעה 23:59

Assignment #2

In this assignment you will have to:

1. Use **Docker Compose** to create an application that is built from 5 services:
   - 2 instances of the pet-store service you did for assignment #1,
   - One additional services you will build: a *pet-order* service,
   - A database service and a reverse-proxy service (NGINX) that you download from DockerHub.
2. The *pet-store* and the *pet-order* services must be **persistent** (data stored in a database)
   - In this presentation I use MongoDB, but you can use another DB if you want.
   - This is the one area that you must update your previous pet-store service – to be persistent.
3. Use Docker Compose to **restart** the pet-store and pet-order 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. Implement **load balancing** for the pet-order service

Two pet-store service instances

You run two different instances of the pet-store service, pet-store1 and pet-store2.

- These represent two different stores with the same owner (a pet-store chain). Each instance can have different pet-types and different pets.
- Each instance is independent and will listen at a different port on the host.
- Each instance of the service must be able to resume to function after a failure
  - Data stored in the database will be available after a crash
- Assignment 1 should be modified to make it persistent (use a DB to store the data). If there were errors in assignment 1, you need to fix them.

pet-order service

The pet-order service provides two basic functionalities.

- It allows customers to purchase a pet.
- It allows the pet-stores' owner to view all purchase transactions.

This service has these resource:

/purchases

/transactions

JSON for a purchase

```
{
  “purchaser”: string,
  “pet-type”: string,
  “store”: number,
       // the store field is optional.
       // If supplied, it equals 1 or 2.
  “pet-name”: string,
     // pet-name is optional.  Can only be
     // supplied if store is supplied.
  “purchase-id”:  string
}
```

Example purchase object

```
{
  “purchaser”: “John Jones”,
  “pet-type”: “Poodle”,
  “store”: 2,
  “pet-name”: “Jamie”,
  “purchase-id”: “4763”
}
```

Requests on the /purchases resource by customers

- **POST**: The POST request provides a JSON purchase object as the payload. It does not contain the "purchase-id" field but must contain "purchaser" & "pet-type" fields.
- The server returns the status code 201 with the JSON purchase object including the "purchase-id" field (a string). **The "store" and "pet-name" fields are returned even if they were not included in the request**.
- Possible error status codes returned: 400 and 415. If there does not exist a pet-type of that type then 400 is returned with the message: {"error": "No pet of this type is available"}.

When receiving a request, the server must:

1. Check if a pet is available according to the request. (See next slide for details).
2. If not, then return 400 with an error message. Otherwise, record the pet chosen, the store it is from, and pet-name.
3. DELETE the chosen pet from the pet-store.
4. Store the transaction JSON in the transactions Mongo collection with all fields.
5. Return the purchase JSON object to the customer.

Note on choosing a pet of a given type

Choosing a pet

- If the store and pet-name are indicated in the request, then the pet of that name must exist in the given store, and that pet is chosen.
- If a store is indicated in the request but no pet-name is given, then there must be at least one pet of the given pet-type in the store. Choose a random pet from the ones available.
- If no store is indicated in the request, then there must be at least one pet of that type in one of the stores. Choose a random pet from the ones available.
- If no pet meets these criteria then return 400.

Finding the id of a given pet-type

- You are given the pet-type but **not** its id. To get its id you need to contact the pet-store and retrieve the appropriate information to find out its id.

JSON for a transaction

```
{
  “purchaser”: string,
   “pet-type”: string,
   “store”: number,
   “purchase-id”:  string,
}
```

Example transaction object

```
{
   “purchaser”: “John Jones”,
   “pet-type”: “Poodle”,
   “store”: 2,
   “purchase-id”: “4763”
}
```

Requests on /transactions by owner

There is no POST request for transactions.

There is only a GET request by the pet-store owner.

The data for the GET is obtained from the MongoDB in the transactions collection.

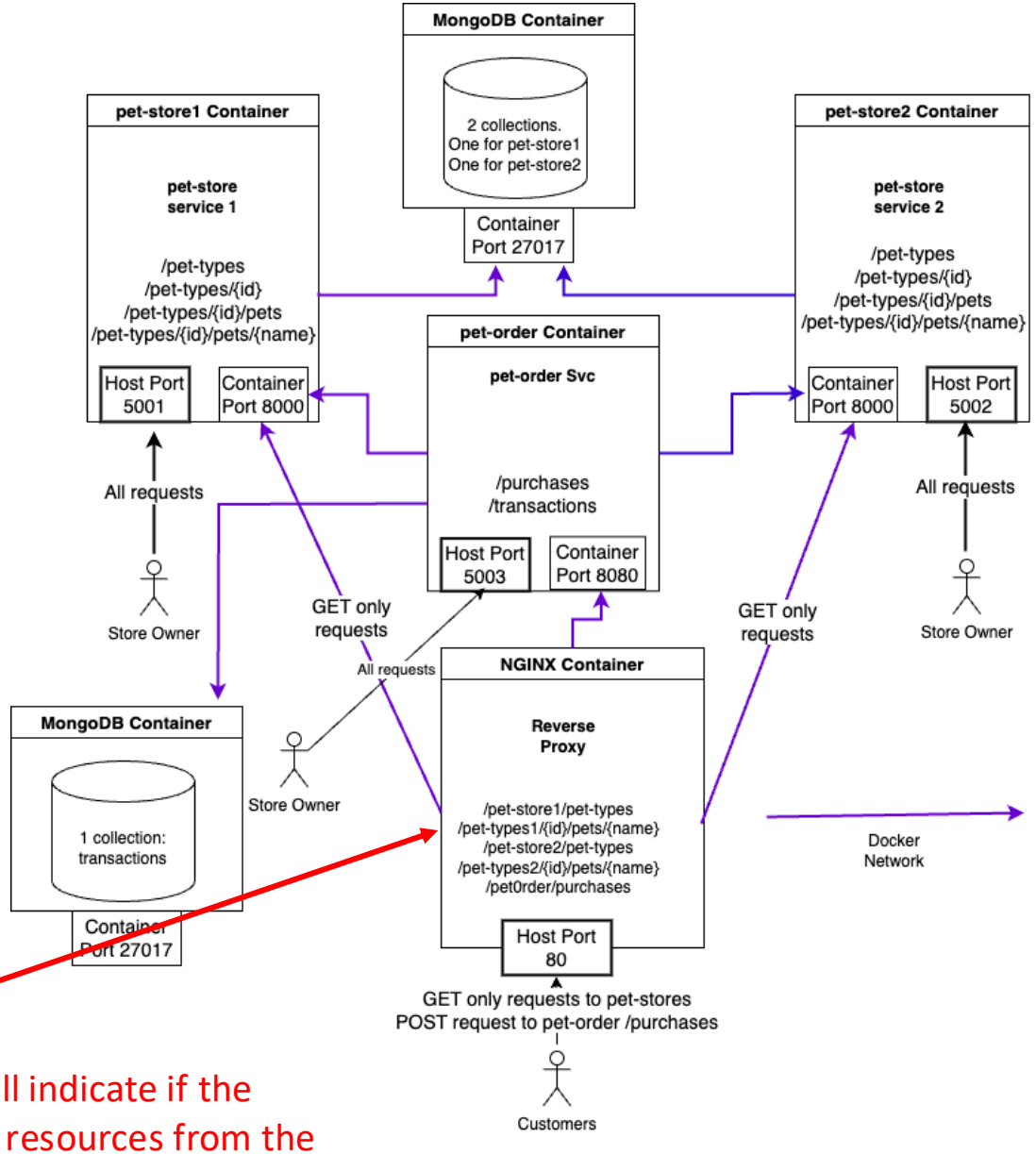
The GET is only allowed if the header contains:
“OwnerPC”: “LovesPetsL2M3n4”

- **GET request on /transactions resource**: The server returns a JSON array of transactions with status code 200.
- The only error status code is 401 “unauthorized”. This occurs if the header does not contain “OwnerPC: LovesPetsL2M3n4!”
- You need to support query strings of the form “**<field-name>=<value>”,** where <field-name> is one of the fields of a transaction JSON object.

Example query strings:
“store=1”
“pet-type=poodle”

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 pet-store1 and pet-stores2 and pet-order microservices



Note that the request to a pet-store resource will indicate if the resource is in pet-store 1 or pet-store 2. All the resources from the pet-store service can be requested.

NGINX resources

When sending requests on the NGINX service, here are the resources that can be requested

From pet-store1:

- /pet-types1
- /pet-types1/{id}
- /pet-types1/{id}/pets
- /pet-types1/{id}/pets/{name}

From pet-store2:

- /pet-types2
- /pet-types2/{id}
- /pet-types2/{id}/pets
- /pet-types2/{id}/pets/{name}

From pet-order service a POST request to the resource:

- /purchases

Summary of host ports

Service	Host Port
pet-store1	5001
pet-store2	5002
pet-order	5003
NGINX	80

Note that in the architecture I also list container ports. But the TA will not issue any requests on container ports – only your implementation does so. So you free to give them other port numbers.

Why docker containers communicate on docker ports

- A docker container runs on a docker network. *localhost* for that docker container refers to the loopback interface **of that individual container**, not the host machine.

Instead docker containers should communicate with each other using their **container** ports.

- Docker containers receive their own IP addresses within the docker network they are running on.
- Docker containers must communicate with each other using their service names (provided in Docker compose) and their container ports (also given in Docker compose).

How to invoke a container API from another container

Assume we have two services, A-svc and B-svc, specified in Docker compose.

- 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?

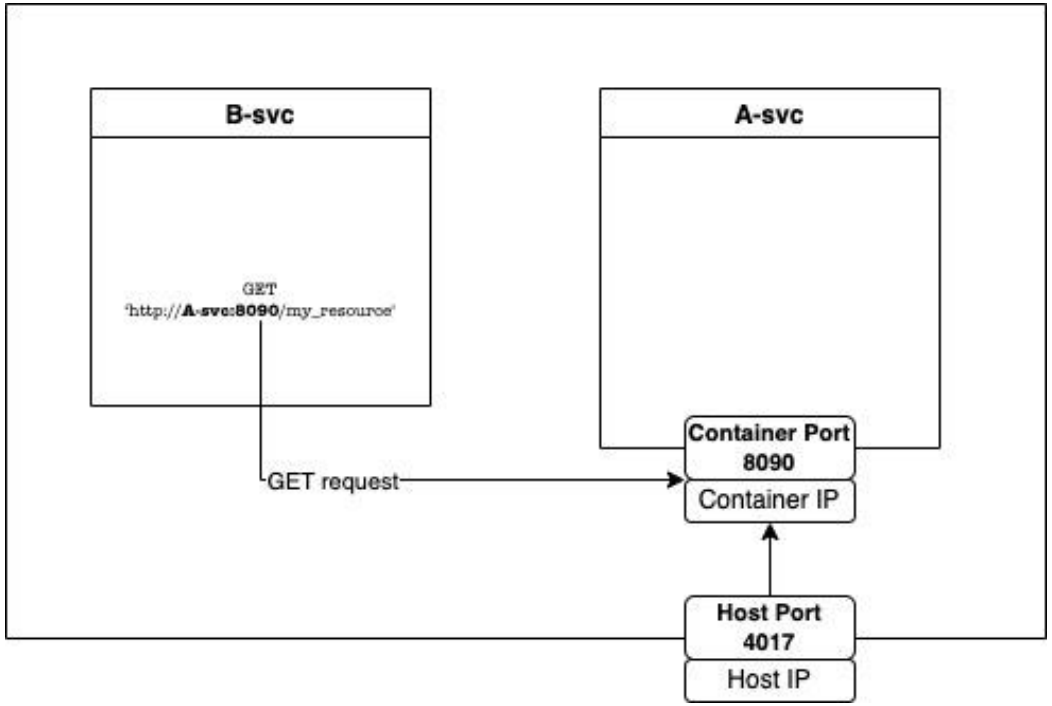
```
version: '3'  # version of compose format

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'
```

- It invokes 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.



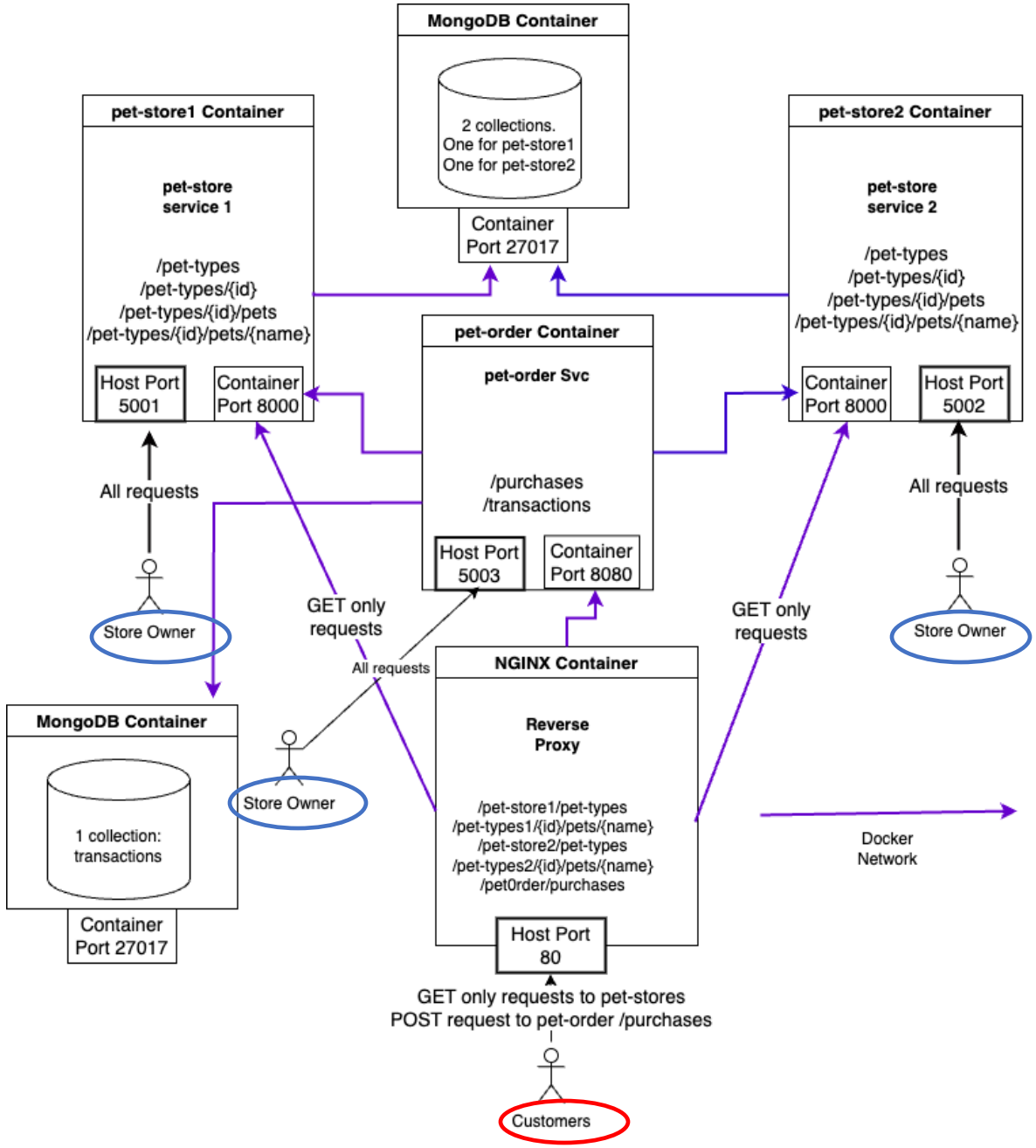
How to invoke a container API from another container (cont)

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

    ...
```

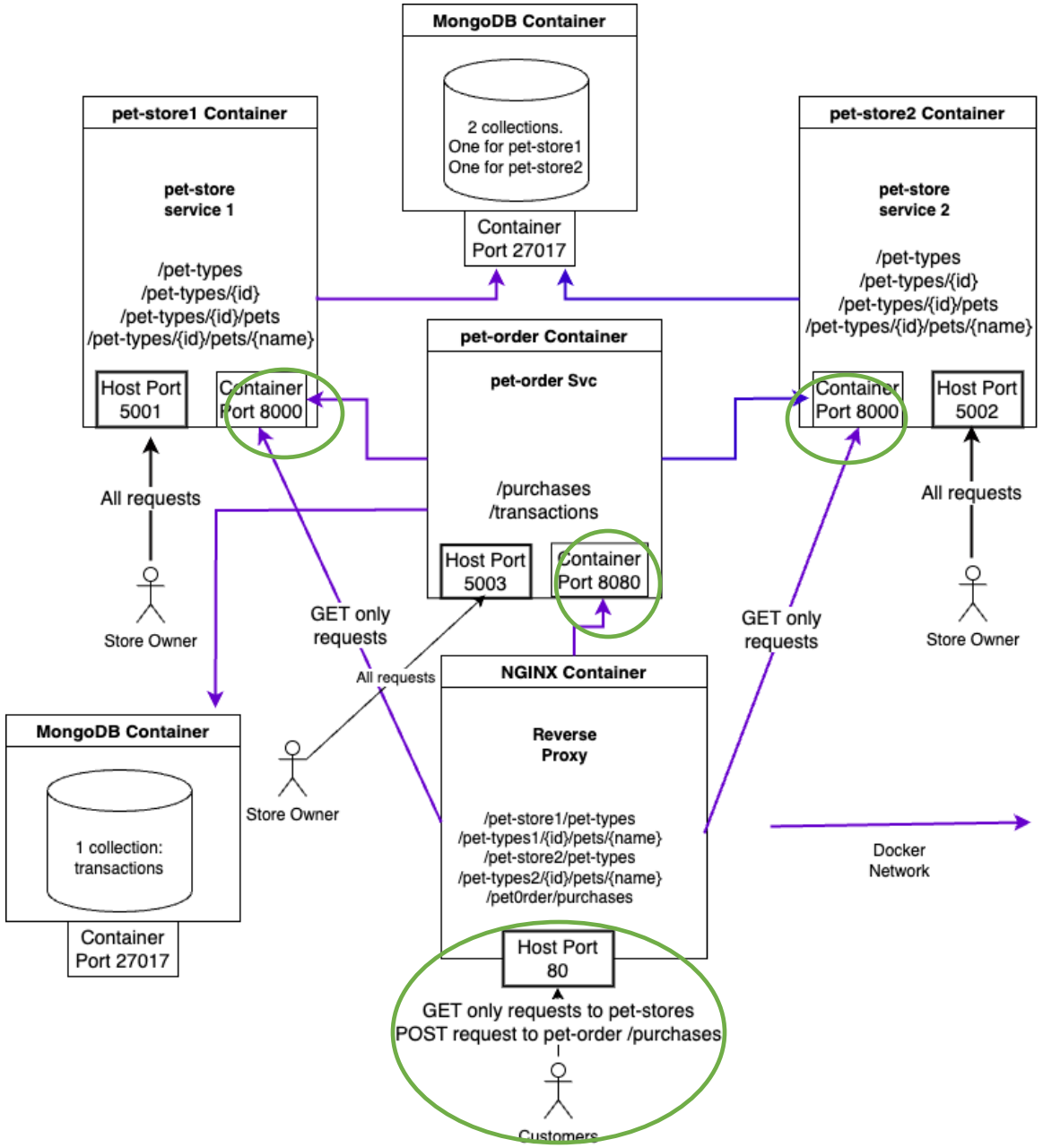
Access rights

- We have 2 *roles* accessing the application: the owner and customers.
- The owner can directly make requests to the pet-store1 & pet-store2 svcs, including POST requests. He can perform GET requests on /transactions in the pet-order svc.
- Customers can only GET information from the pet-store svcs and can POST to the pet-order svc (/purchases). They must go through the NGINX svc to issue these requests.



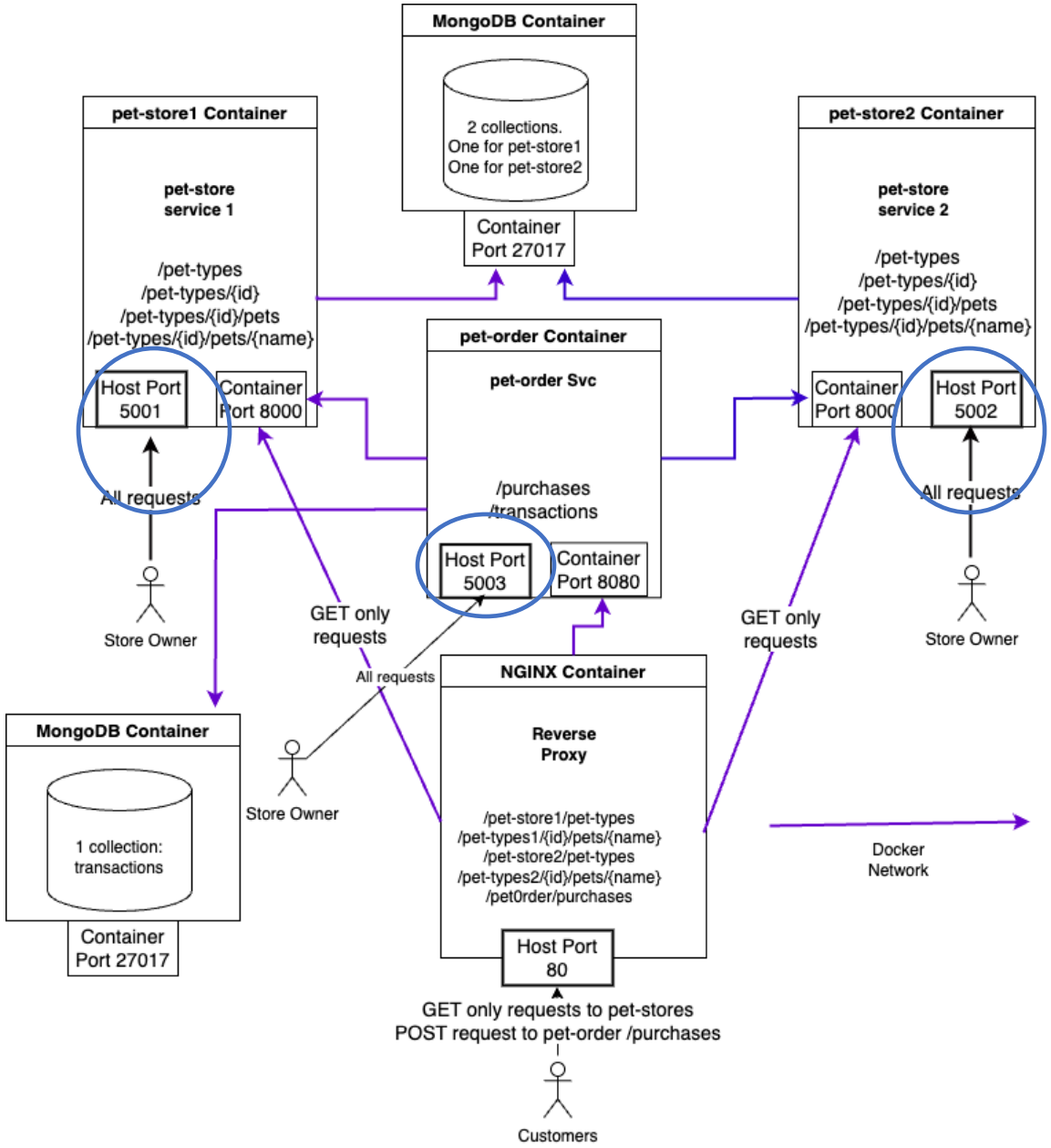
NGINX

- Configure the reverse proxy (NGINX) as in the diagram
- GET requests on the host port 80 for resources in the pet-store1 & pet-store2 svcs get forwarded as requests to the pet-store1 & pet-store2 svcs respectively.
  - Note that NGINX will only accept GET requests for the resources in these svcs. Other requests will be rejected.
- POST requests on the host port 80 for the /purchases resource of the pet-order svc get forwarded as requests to the pet-order svc.
- If other requests are sent to NGINX, it should respond with the 403 Forbidden message (this is the default).



Owner requests

- Requests on host ports 5001, 5002 and 5003 do **not** go through the reverse-proxy.
  - All requests (GET,POST,PUT…) on the pet-store service resources are permitted on 5001 and 5002. No header containing the “OwnerPC” is required (as the svc is the same as in assignment #1).
  - GET requests on the /transactions resources of the pet-order svc (host port 5003) requires the header to contain the “OwnerPC” as described previously.



Docker Compose restart

Your pet-stores and pet-orders containers need to be able to automatically restart on failure. This is accomplished by using the Docker Compose restart instruction, as illustrated in class in the high availability lecture.

Testing Docker Compose restart

In order to test that your pet-store and pet-order containers automatically restart on failure, these services need to implement the following /kill request.

```
@app.route('/kill', methods=['GET'])
def kill_container():
    os._exit(1)
```

Upon executing the `GET /kill` request, the service will execute os._exit(1). This command will kill the main process and cause the container to fail.

Load balancing

You only need to implement load balancing for pet-order service.

- You should have **2 instances** of the **pet-order** service.
- You should use a weighted round robbin scheduling policy, as we discussed in class (see NGINX slides).
- For every 3 requests that the **first** instance of **pet-order** service receives, the **second** instance of the **pet-order** service should receive 1 request. Saying the same thing in different words: the first **pet-order** service instance receives 3 times more the number of requests than the second **pet-order** service instance receives.
- Note that the load balancing is only for **pet-order** service.

The second instance of the **pet-order svc** need not be given a host port in the docker-compose.yml as it cannot be reached from the host. Only NGINX commumicates to it on the Docker Network. So it only needs to expose its container port.

2 pet-store and 2 pet-order service instances

- Although you will have 2 pet-store and 2 pet-order service instances, they are very different:
  - Each pet-store is an independent service. It will have its own data (different pet-types and different pets**). You have two of them because they manage two different stores.**
  - The two pet-order instances read and write data to the same database collection (same pet-types and same pets). **You have two instance only for load balancing.**
- You can imagine having 4 pet-store instances:
  - For each store, having two instances for load balancing.
  - That would have made the assignment a little more challenging!

Submitting your assignment

When submitting your assignment, please **attach a document listing the team members**. This should be done even if submitting alone.

Submission

- The code you write will need supply all the files required to run your application.
  - This includes code for your services, the Dockerfiles required for your services, the docker-compose.yml, the configuration file for NGINX and any other required files.
- You should submit all of your code in a zip file. If your docker-compose or 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 your assignment works before submitting.** I recommend that you give your zip file to a friend to unzip and issue the Docker Compose build and run commands. I.e., make sure that it will work for the TA (not only on your computer) before submitting.

Grading of the assignment (1)

We will test your code by:

- Running your docker-compose.yml to build the images and containers, and then run the application.
- Issuing REST requests to the different services on the specified host ports and checking that the correct responses are returned.
  - Includes checking query strings work properly.
- Testing that your services automatically restart after failure.
- Testing the persistence of the pet-store and pet-order services, even after they fail and are restarted.
- Testing that NGINX forwards requests properly and denies requests according to the given policies.
- Testing NGINX load balancing implements weighted round robin correctly.

Grading of the assignment (continued)

The TA will issue REST requests on your Docker containers:

- Does the Docker Compose up command work? Does it create the right images and containers?
- Do the containers built from the images run your code without a problem? Are they running at the right IP address?
- Does it process POST, GET, DELETE and PUT requests properly?
- Does it route the requests for different resources to the correct code?
- Does it return the right status codes for each request?
- Does it return the right JSON for each request?
- Does it handle incorrect requests properly, returning the right status codes and JSON error messages?