Faculdade de Engenharia da Universidade do Porto

Sistema de monitorização e simulação de mobilidade

Grupo 4

Afonso Cunha - up202003308
David Blanquett - up201909456
João Fernandes - up202003038
Pedro Carvalho - up202003056

Application Requirements and System Operations

Domain model

- **RQ1**: As a Researcher, I want to create, update and delete a domain model in the form of a graph with the JSON format.
    - AC1.1: Define the type: macroscopic, microscopic, mesoscopic.
    - AC1.2: Define all the topology characteristics.
    - AC1.3: Register in the monitoring system.
- **RQ2**: As a user, I want to read a domain model in the form of a graph with the JSON format.
    - AC2.1: Verify that at least one domain model exists.
    - AC2.2: Register in the monitoring system.
- **RQ3**: As a user, I want to read all existing domain models of a given type (macroscopic, microscopic, mesoscopic).
    - AC3.1: Specify a valid topology type.
    - AC3.2: Register in the monitoring system.
- **RQ4:** As a User, I want to read and copy other user's models
    - AC4.1: User can't delete the other user's model
    - AC4.2: Register that a user accessed another user's model.
    - AC4.3: Register if a user copied another user's model.

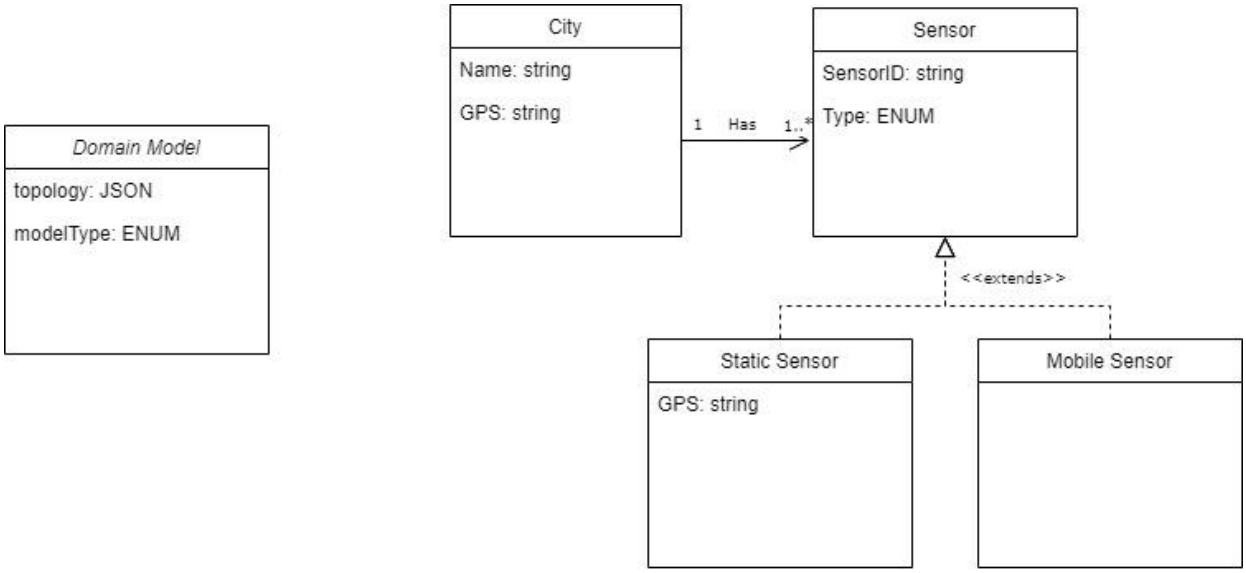
Open question: Should all users be able to update and delete domain models created by others? Or only users that created a domain model can update and/delete it?

City

- **RQ3**: As a Researchers or Service Provider, I want to create, update and delete a city.
    - AC3.1: Define the city characteristics: name and GPS location.
    - AC3.2: Register in the monitoring system.
- **RQ4**: As a Researchers or Service Provider, I want to see all cities.
    - AC4.1: Verify that at least one city exists.
    - AC4.2: Register in the monitoring system.
- **RQ5**: As a Researchers or Service Provider, I want to create, delete and update sensors in a city.
    - AC5.1: Specify the sensor mobility (mobile or static) and respective characteristics: type and description.
    - AC5.2: Define a valid city to insert the sensor.
    - AC5.3: Register in the monitoring system.
- **RQ6**: As a user, I want to see all sensors in a city.
    - AC6.1: Specify an existing city.
    - AC6.2: The city must at least contain one sensor.
    - AC6.3: Register in the monitoring system.
- **RQ7**: As a user, I want to see all sensors in a city filtered by type and mobility.

    - AC7.1: Specify an existing city.
    - AC7.2: Define if it is static or mobile.
    - AC7.3: Select a valid sensor type (e.g. temperature, speed).
    - AC7.4: The city must at least contain one sensor.
    - AC7.5: Register in the monitoring system.
- **RQ8**: As a user, I want to see all fixed sensors in a given radius range.
    - AC8.1: Specify a valid value for the radius.
    - AC8.2: Register in the monitoring system.
- **RQ9**: As a user, I want to see all fixed sensors in a given radius range filtered by type.
    - AC9.3: Specify a valid value for the radius.
    - AC9.2: Select a valid sensor type.
    - AC9.3: The city must at least contain one static sensor.
    - AC9.4: Register in the monitoring system.

Domain Model UML Diagram

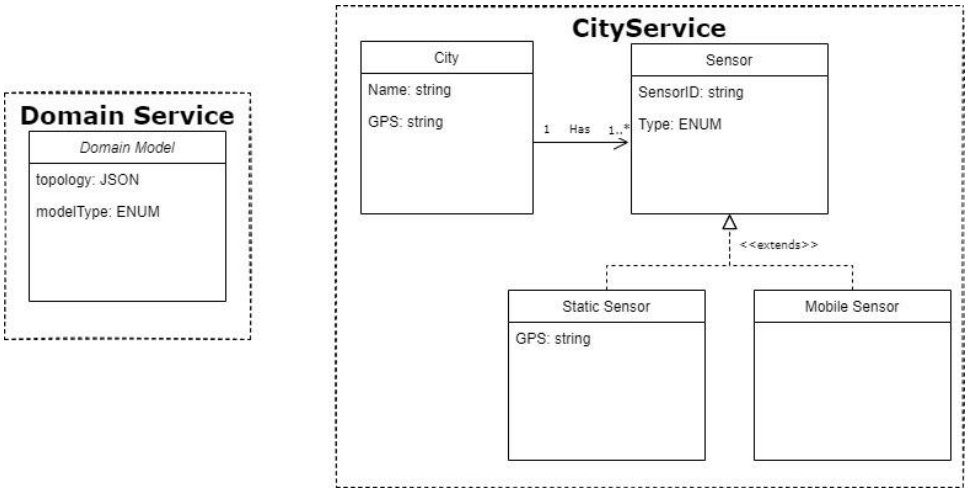


Services

When splitting our domain model into microservices the first problem is identifying microservice boundaries and the “right” size for each microservice. According to recent bibliography to map a domain model into microservices we need toto Identity aggregates in our model. A well-designed aggregate should be concerned with the following properties:

- An aggregate should be derived from business requirements, rather than technical
- And aggregate should have high functional cohesion.
- An aggregate is a boundary of persistence.
- Aggregates should be loosely coupled.

Applying this rules into our domain, we can cluster the functionalities into two aggregates. One regarding the “Domain” and other regarding the “City”. This led us to considering two services, the “Domain Service” and the “City Service”, as we can see from Figure XXXXX.



In order to validate our design we should consider the following properties:

- Each service has a single responsibility.
    - “Domain Service” is responsible for managing the domain
    - “City Service” is responsible for managing the city
- There are no chatty calls between services. If splitting functionality into two services causes them to be overly chatty, it may be a symptom that these functions belong in the same service.
    - “Domain Service” and “City Service” don't need to communicate with each other.
- Each service is small enough that it can be built by a small team working independently.
    - “Domain Service” and “City Service” can be divided.
- There are no inter-dependencies that will require two or more services to be deployed in lock-step. It should always be possible to deploy a service without redeploying any other services.
    - “Domain Service” and “City Service” don’t have direct coupling.
- Services are not tightly coupled, and can evolve independently.

    - “Domain Service” and “City Service” are not tightly coupled, and can evolve independently.
- Your service boundaries will not create problems with data consistency or integrity. Sometimes it's important to maintain data consistency by putting functionality into a single microservice. That said, consider whether you really need strong consistency. There are strategies for addressing eventual consistency in a distributed system, and the benefits of decomposing services often outweigh the challenges of managing eventual consistency.
    - “Domain Service” and “City Service” don’t have direct coupling.

To conclude the division, we also like to add that this type of approach is supposed to be an iterative and incremental process, and only when the system is more mature, we can validate if the division is well suited. This can be further validated using dynamic/static analysis of service calls.

Technologies

We will devide this section into two parts. One referring the technologies behind “Domain Service” and other referring “City Service”. Both service will provide a REST API in order to exchange information with external components.

Domain service was going to be developed using the stack the MEN stack, which is stands for MongoDB as Database, Express and Node.JS as backend. But due to technology researching we decided to choose a Microsoft based approach, using .Net Core 6.

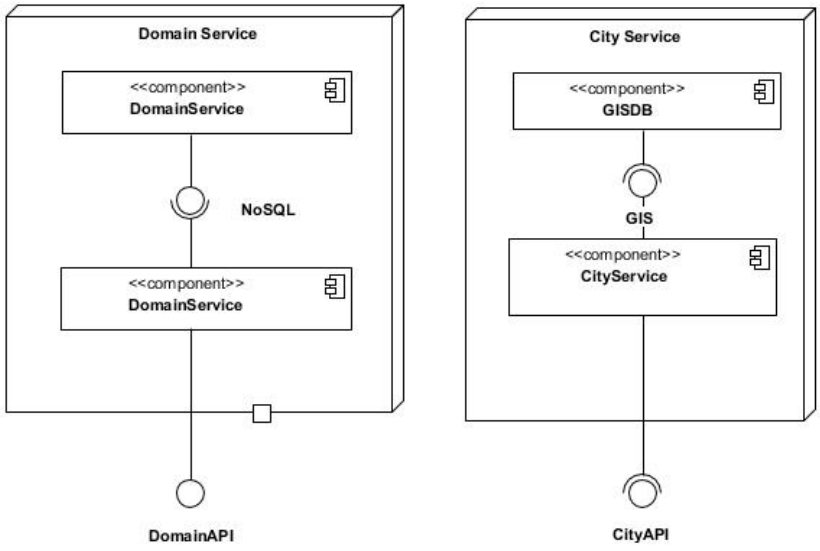
Regarding deploy we decided to use Kubernetes, with the following metadata:

JwtParser.cs | DomainModelRepo.cs | DomainService: Publish | domainService-depl.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: domainmodels-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: domainmodelservice
  template:
    metadata:
      labels:
        app: domainmodelservice
    spec:
      containers:
        - name: domainmodelservice
          image: dasb98/domainmodelservice:latest
---
apiVersion: v1
kind: Service
metadata:
  name: domainmodels-clusterip-srv
spec:
  type: ClusterIP
  selector:
    app: domainmodelservice
  ports:
  - name: domainmodelservice
    protocol: TCP
    port: 80
    targetPort: 80
  - name: plafromgrpc
    protocol: TCP
    port: 666
    targetPort: 666
```

City Service is going to be developed in python and using a GIS Database, the design decision behind this is going to be described in the following section.

The physical implementation can be described by the following picture.



Design Decisions

- How to save the graph?

Adjacency List: An Adjacency list is an array consisting of the address of all the linked lists. The first node of the linked list represents the vertex and the remaining lists connected to this node represents the vertices to which this node is connected. This representation can also be used to represent a weighted graph. The linked list can slightly be changed to even store the weight of the edge.

Adjacency Matrix: Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

As discussed during class we will use a **Adjacency List.**

- Databases

Given the type of data that the *Domain Service* will be storing, a NoSQL database is an adequate choice, due to high-availability and scalability concerns. The *City Service* requires storing GPS coordinates; thus, a spatial database is recommended in order to store the geographical information. The project team decided to choose PostgreSQL for the *Domain Service* and PostgreSQL with the PostGIS extension (to provide spatial support) for the *City Service.*

- Inter-service communication

Inter-service communication will be achieved using REST (Representational State Transfer) using the HTTP protocol. The communication will have a maturity level of 2 according to the Richardson Maturity Model, including resource URIs and HTTP Verbs. Given the type of queries that the *Domain Service* and the *City Service* are expected to receive, asynchronous communication using a messaging system such as Kafka should not be required.

Patterns

The services described in this document use multiple patterns that are frequently identified in software projects in order to ensure important qualities, namely resiliency and observability.

The City service uses the following patterns:

- Timeout (resiliency pattern): the service will stop all requests that take more than 30 seconds to complete, in order to prevent failures from other services to cause a failure in the City service.
- Load balancing (resiliency pattern): the server infrastructure will distribute incoming requests across multiple instances capable of fulfilling them. The load balancer is part of the infrastructure used for the deployment of the service: Heroku.
- Health check API pattern (observability pattern): the service exposes a health check API endpoint, available at GET /health, which returns the status code 200 when the service is working as expected. This endpoint can be used to periodically check the health of the service.
- Logging (observability pattern): for every request, the service logs the log type (information or warning), the operation type (read, patch, update, create, delete, other), the response status code, the user, and the endpoint requested.

The Domain Model Service uses the following patterns:

- Timeout (resiliency): the service will stop all requests that take more than a determinate amount to complete, in order to prevent failures from other services to cause a failure in the Domain Model Service.
- Load balancing (resiliency pattern): We accomplish this by deploying into Azure Cloud. We also tried to do this using Kubernetes, but it was causing some inconsistency in the data.
- Always on (resiliency pattern): As we can see in the ANNEX, Kubernetes manages the infrastructure so he can keep his metadata properties.
- Health check API pattern (observability pattern): the service exposes a health check API endpoint, available at GET /health, which returns the status code 200 when the service is working as expected, and 503 is the service is down. When returning the 200 status code, in the message payload can be seen the performance status of the application (based on CPU usage). This endpoint can be used to periodically check the health of the service.
- Logging (observability pattern): for every request, the service logs the log type (information or warning), the operation type (read, patch, update, create, delete, other), the response status code, the user, and the endpoint requested. To accomplish this we implement a custom kafka producer that handles this async.

Security:

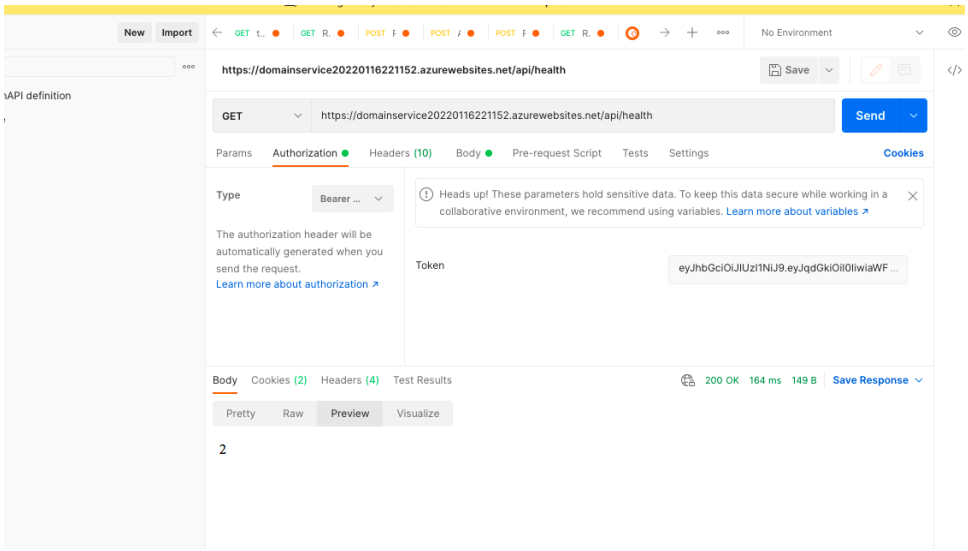
JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed by a third party (Auth0). JWTs are signed using a secret (with the HMAC algorithm), which serves as proof that the payload was not violated, since its signed and cannot be changed without the Auth0 Entity or the private key.

GetRolesFromToken is a function that receives the user's token in string format and retrieves a payload with the user's roles also in string format. If the token is incorrect it causes an exception and shows a error message.

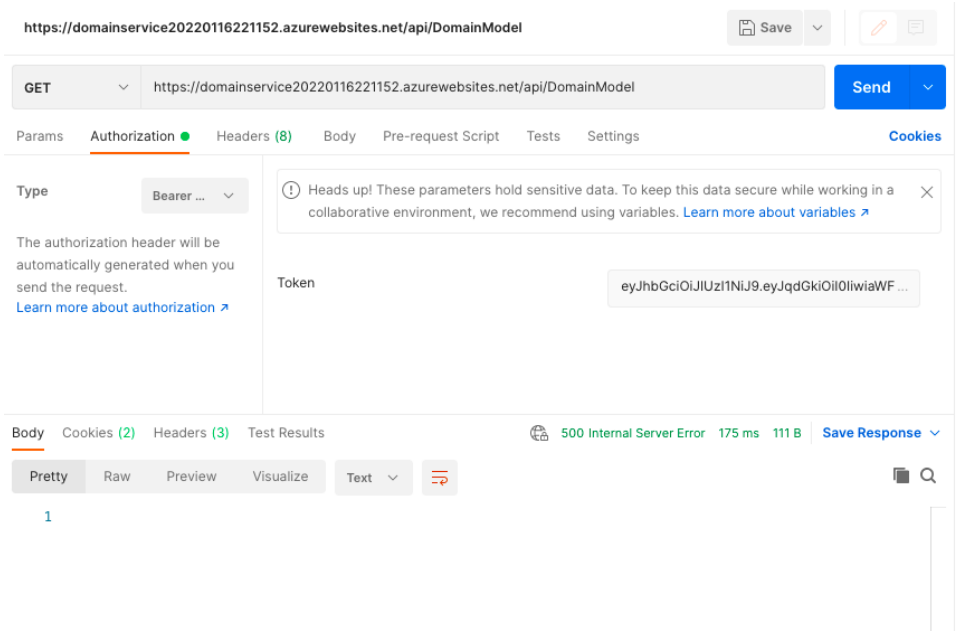
Tests:

With Token:

API Health:

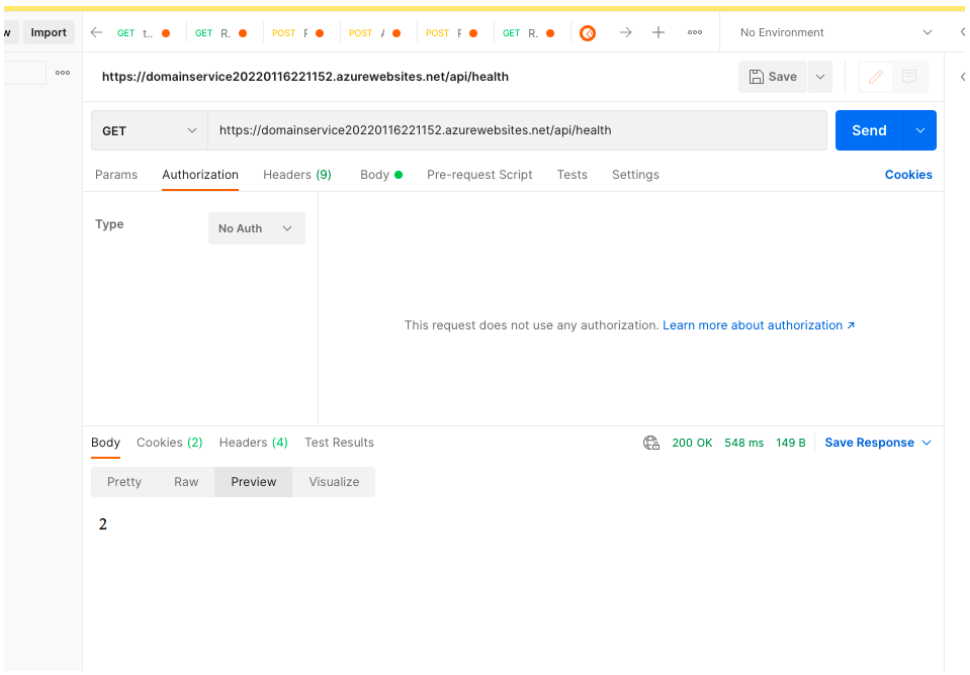


Get Domain Model:



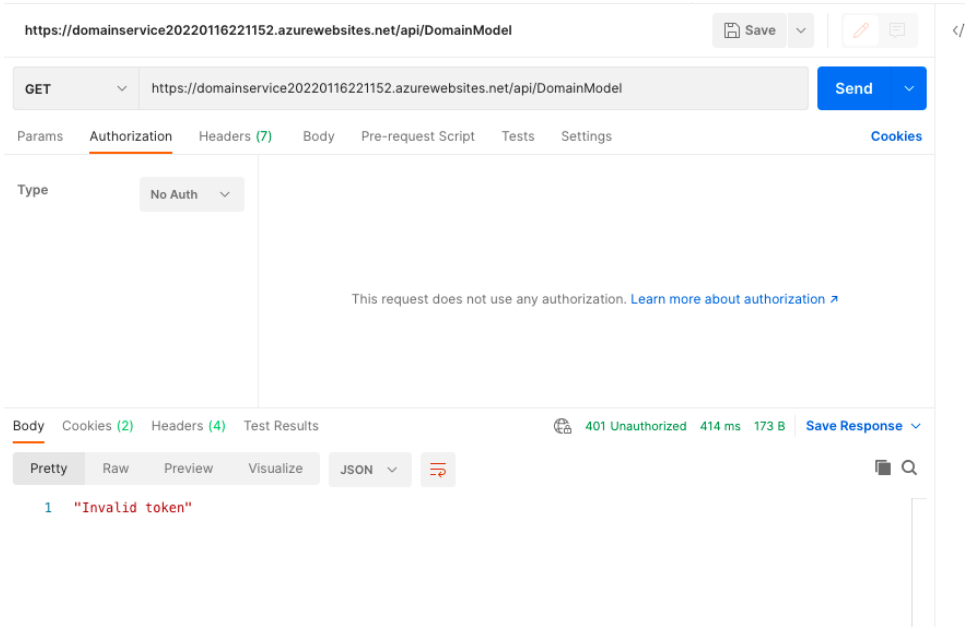
Without Token:

API Health:



API Health endpoint works with and without the token.

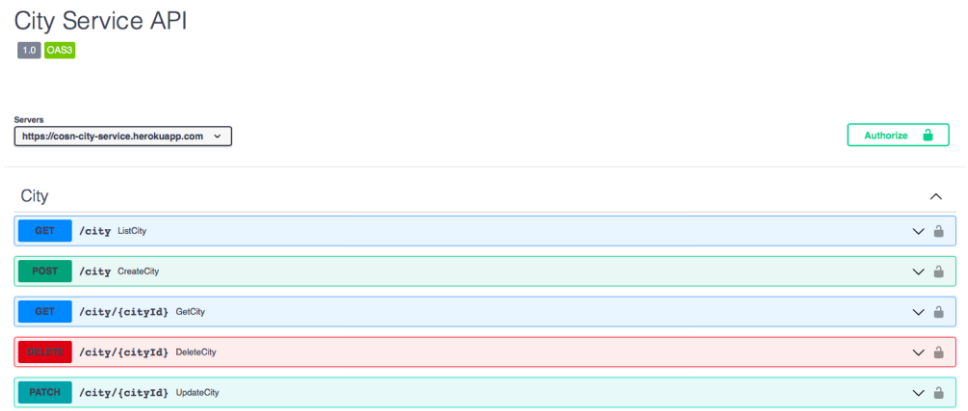
Get Domain Model:



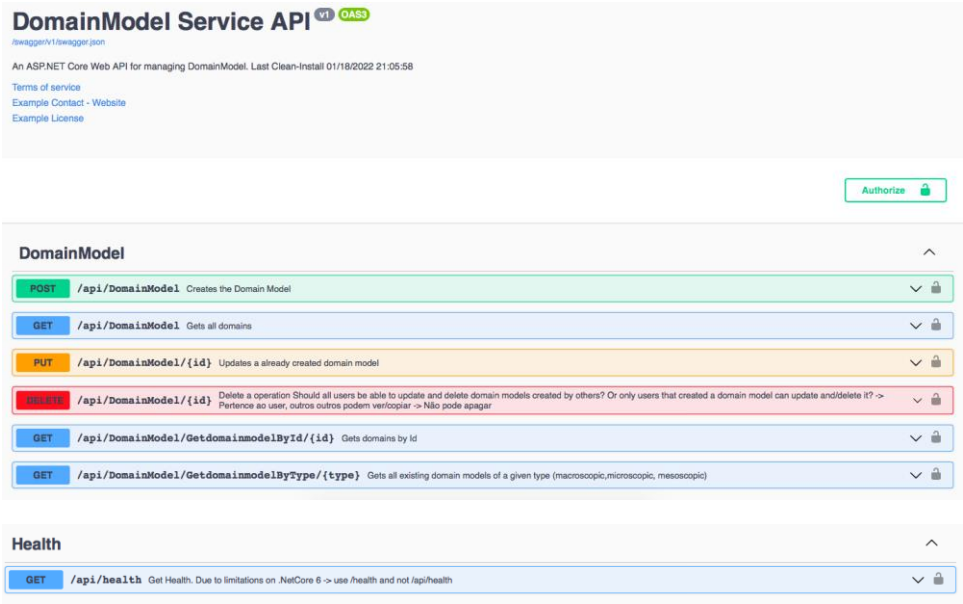
Service APIs and collaborations

The services APIs are documented according to the OpenAPI specification, available on the following links:

City Service: https://app.swaggerhub.com/apis/officialpedrocarvalho/CityServiceAPI/1.0



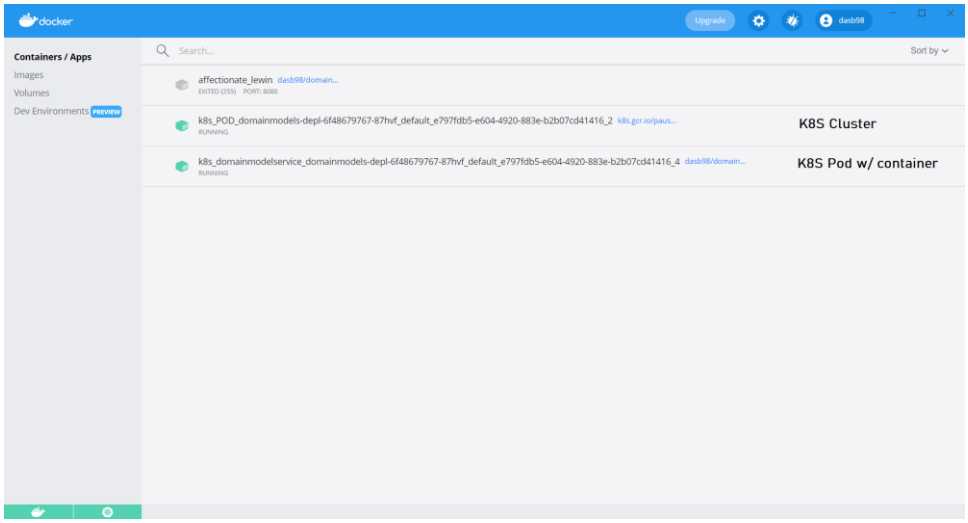
Domain Service: https://domainservice20220116221152.azurewebsites.net/index.html



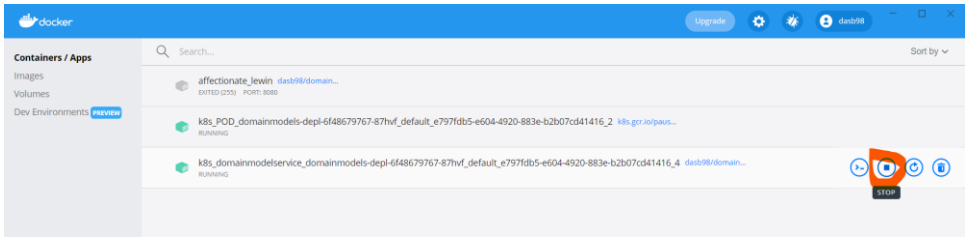
ANNEX:

Resiliency using Kaburnetes:

Initial state:



Delete pod:



After a few seconds: