Design and Implementation of an API Gateway for Microservices on Kubernetes

My project is a **microservices** architecture that uses an API Gateway. It has been developed in **Kubernetes**, using **Minikube**. Minikube offers a single Kubernetes cluster, composed of a single node (minikube). On this node I installed and configured the entire architecture of my service, which, in a real case scenario, should be deployed on a much higher number of nodes.

At the application level there is an **e-commerce service** which is divided into five microservices:

- Authentication microservice;
- Authorization microservice;
- Products microservice;
- Orders microservice;
- Database microservice.

These microservices (except for the Database microservice) are simple web servers configured using **Python Flask**.

All clients' requests are received by the **API Gateway**, which forwards the requests to the correct microservice. It uses the authentication microservice to authenticate the requests that need authentication and the authorization microservice to authorize the requests that need authorization. If a request that needs both authentication and authorization will be validated by both the corresponding microservices, it will be forwarded to the correct microservice. This procedure is executed transparently to the client. The adopted API Gateway is configured with **Traefik**.

The **Authentication microservice** is publicly available to the users (or clients) and is used for login, registration and authentication of all the requests. This means that the API Gateway routes all the requests it receives (at the least the ones that require authentication) to this microservice, in order to validate (i.e. authenticate) the requests. The other microservices don't have to implement authentication (i.e. checking cookies) to know the identity of the users, they can simply read the "**X-User-ID**" header that the authentication microservice has attached to all the authenticated requests. If the authentication microservice can't authenticate a request (invalid, expired or absent JWT), an error (HTTP status code: 401) is returned to the API gateway and this error is then forwarded to the client. All requests except the ones for */authentication/login* and */authentication/registration* endpoints require authentication. The exposed endpoints are:

- ***/authentication/login*** : allows a user to login;
- ***/authentication/logout*** : allows a user to logout (its cookie will be blacklisted);
- ***/authentication/registration*** : allows a user to registrate to the service;
- ***/authentication/whoami*** : allows to a user to get its username;
- ***/validate*** : allows the API gateway to authenticate requests (this endpoint is not publicly available).

The **Authorization microservice** is **not** publicly available, but it is used by the API gateway to authorize some requests. If the authorization microservice does not authorize a request (returns HTTP status code 401 to the gateway), an error is then forwarded to the client. In this application there is a single endpoint which requires authorization (*/products/admin*). The only interesting endpoint of this microservice is ***/validate***, which is used by the API gateway to authorize certain requests.

The **Products microservice** is publicly available and contains the following endpoints:

- ***/products*** : returns all available products;
- ***/products/admin*** : returns a product that only the *admin* can see (this endpoint requires authorization);
- ***/products/{id}/buy*** : allows to buy a product (whose id is specified in the path).

The **Order microservice** is publicly available and contains the following endpoints:

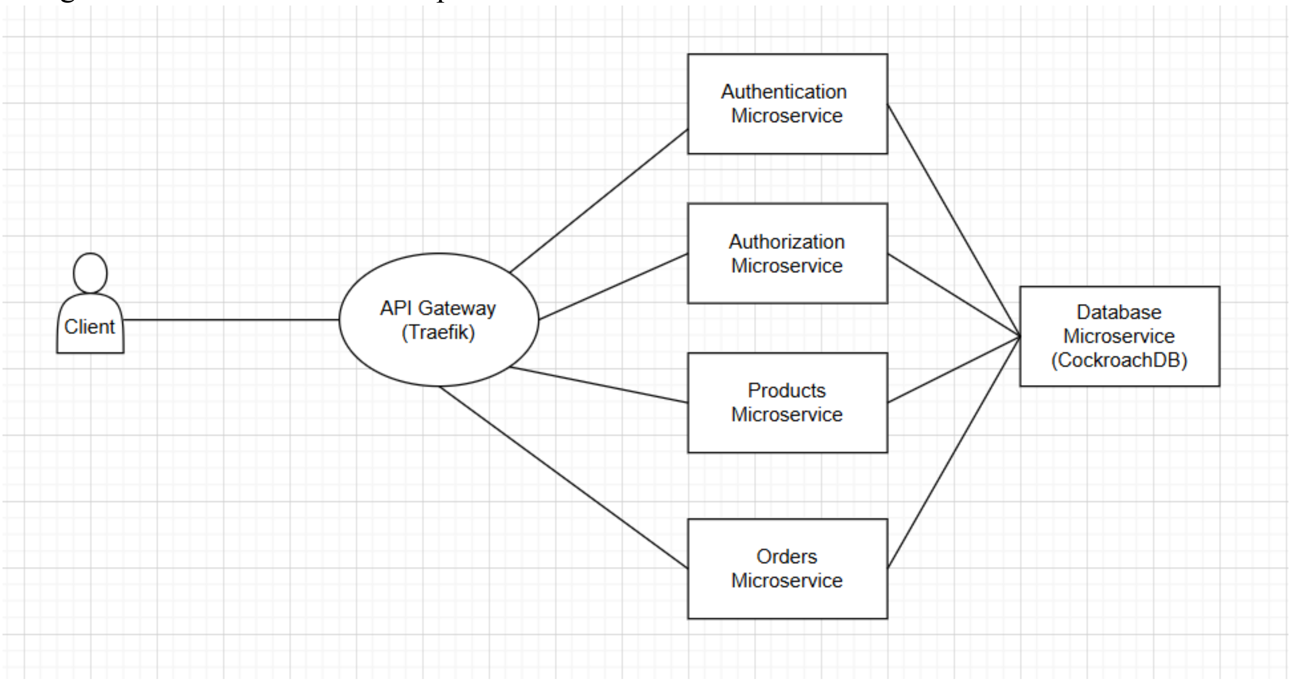
- ***/orders*** : returns all the orders of a user;
- ***/microservice/orders/add*** : allows to add an order (this endpoint is not publicly reachable and is accessed only by the Products Microservice).

The **Database microservice** is **not** publicly available and handles a distributed SQL database. Only the other microservices use this hidden microservice. This microservice in based on **CockroachDB** distributed DBMS, which manages the SQL database and automatically guarantees all the non-functional properties a database should have in a distributed environment (fault tolerance, high availability, security, scaling, sharding, consistency and so on).

This e-commerce application allows a user to:

- Login (*/authentication/login*)
- Registrate (*/authentication/registration*)
- Know its username (*/authentication/whoami*)
- Get the available products (*/products*)
- Buy a product (*/products/{id}/buy*)
- Check its orders (*/orders*)
- Retrieve a special product if the user is the *admin* (*/products/admin*)

The general architecture can be represented like this:


Authentication Microservice
Authorization Microservice
Client
API Gateway (Traefik)
Database Microservice (CockroachDB)
Products Microservice
Orders Microservice

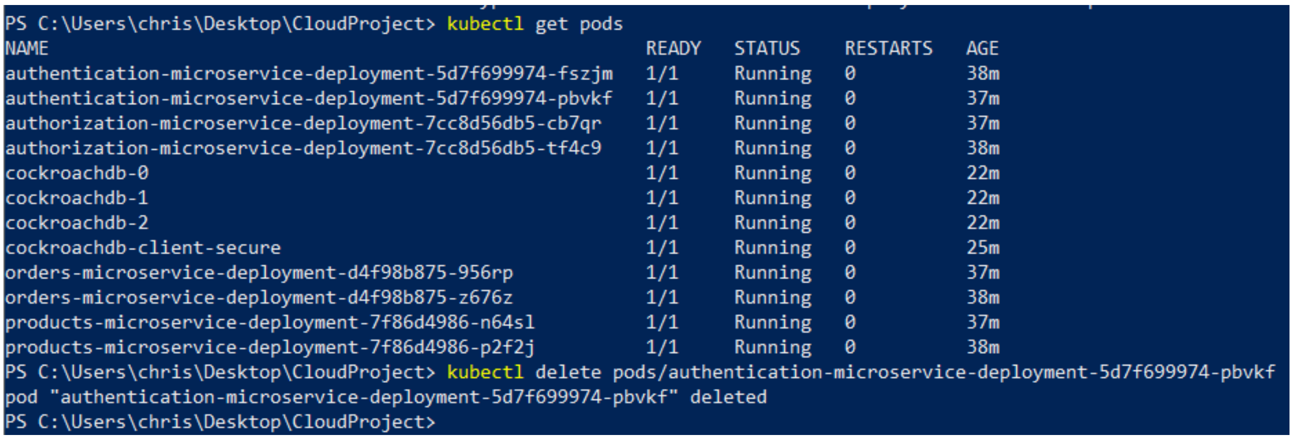
Let’s focus on the non-functional properties of this architecture.

Fault Tolerance (and High Availability)

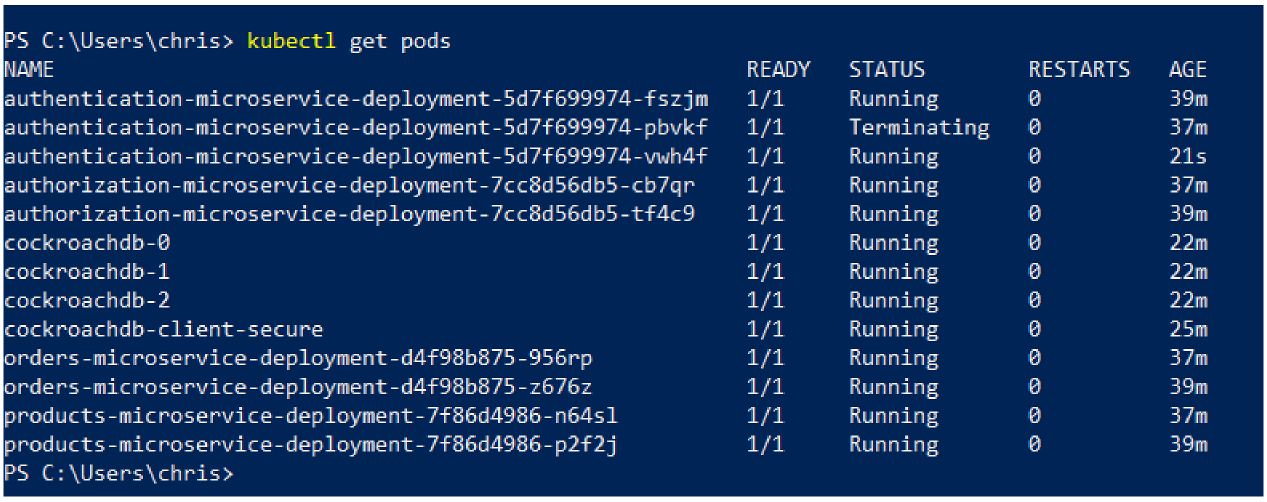
Fault tolerance is the capability to mask the occurrence of a fault (a pod or a container crash for example) within the architecture. Being fault tolerant is strongly related to being dependable. Dependability covers multiple requirements, including **high availability**. In the described architecture fault tolerance is guaranteed by the fact that there are always at least two replicas of each microservices’ pod. Each time a pod faults (i.e. crashes), Kubernetes restores that pod. Usually it’s better to have at least two replicas, but the minimum (or exact or even the maximum) number of replicas can be configured.

In this case **Horizontal Pod Autoscalers** (HPAs) have been configured for each microservice and the API gateway, each of which defines a minimum number of replicas equals to **2** and a maximum number of replicas equals to **3**. This means that even if a pod is deleted, a new one will be immediately started. This configuration is a simple configuration (a minimal one, in order to avoid physical resources’ problems), but in a real case scenario the API gateway and microservices like the Authentication microservice deserve a much higher number of replicas than the other microservices (since they are involved in a higher number of requests).

Here is an example with the authentication microservice, before and after the deletion of pod *authentication-microservice-deployment-5d7f699974-pbvkf* (the second pod in the list):



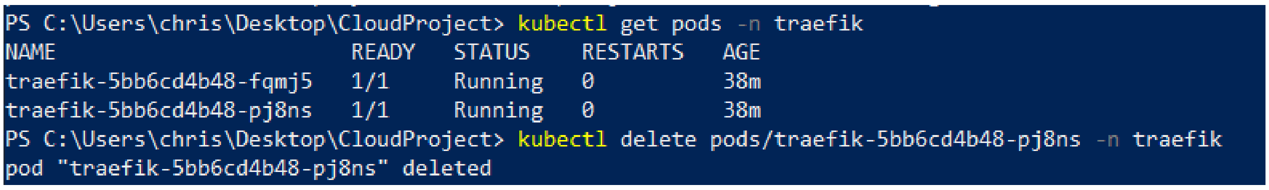
```
PS C:\Users\chris\Desktop\CloudProject> kubectl get pods
NAME                                                READY   STATUS    RESTARTS   AGE
authentication-microservice-deployment-5d7f699974-fszjm   1/1     Running   0          38m
authentication-microservice-deployment-5d7f699974-pbvkf   1/1     Running   0          37m
authorization-microservice-deployment-7cc8d56db5-cb7qr    1/1     Running   0          37m
authorization-microservice-deployment-7cc8d56db5-tf4c9    1/1     Running   0          38m
cockroachdb-0                                             1/1     Running   0          22m
cockroachdb-1                                             1/1     Running   0          22m
cockroachdb-2                                             1/1     Running   0          22m
cockroachdb-client-secure                                 1/1     Running   0          25m
orders-microservice-deployment-d4f98b875-956rp            1/1     Running   0          37m
orders-microservice-deployment-d4f98b875-z676z            1/1     Running   0          38m
products-microservice-deployment-7f86d4986-n64sl          1/1     Running   0          37m
products-microservice-deployment-7f86d4986-p2f2j          1/1     Running   0          38m
PS C:\Users\chris\Desktop\CloudProject> kubectl delete pods/authentication-microservice-deployment-5d7f699974-pbvkf
pod "authentication-microservice-deployment-5d7f699974-pbvkf" deleted
PS C:\Users\chris\Desktop\CloudProject>
```



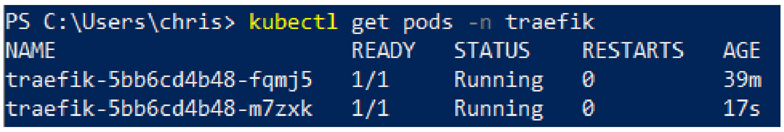
```
PS C:\Users\chris> kubectl get pods
NAME                                                      READY   STATUS        RESTARTS   AGE
authentication-microservice-deployment-5d7f699974-fszjm   1/1     Running       0          39m
authentication-microservice-deployment-5d7f699974-pbvkf   1/1     Terminating   0          37m
authentication-microservice-deployment-5d7f699974-vwh4f   1/1     Running       0          21s
authorization-microservice-deployment-7cc8d56db5-cb7qr    1/1     Running       0          37m
authorization-microservice-deployment-7cc8d56db5-tf4c9    1/1     Running       0          39m
cockroachdb-0                                             1/1     Running       0          22m
cockroachdb-1                                             1/1     Running       0          22m
cockroachdb-2                                             1/1     Running       0          22m
cockroachdb-client-secure                                 1/1     Running       0          25m
orders-microservice-deployment-d4f98b875-956rp            1/1     Running       0          37m
orders-microservice-deployment-d4f98b875-z676z            1/1     Running       0          39m
products-microservice-deployment-7f86d4986-n64sl          1/1     Running       0          37m
products-microservice-deployment-7f86d4986-p2f2j          1/1     Running       0          39m
PS C:\Users\chris>
```

As can be seen, the deleted pod went in *Terminating* status (before being completely removed) and a new one is already in the *Running* status.

Another example with the Traefik API gateway:



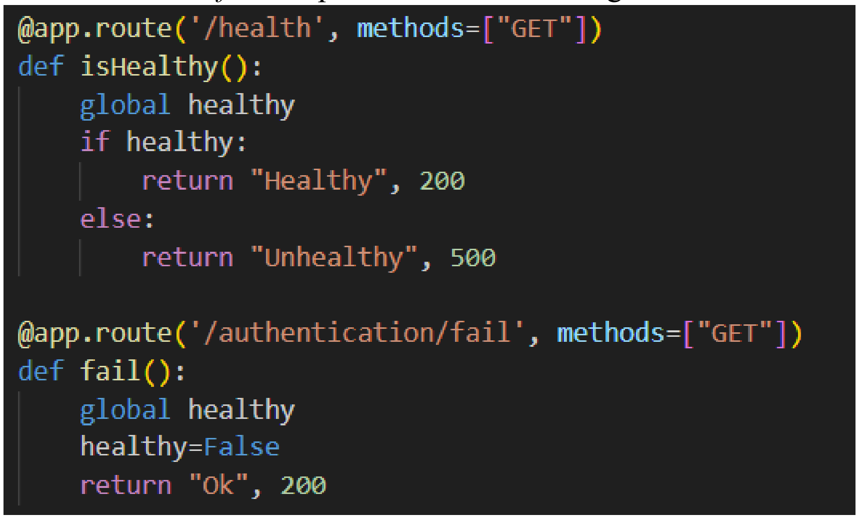
```
PS C:\Users\chris\Desktop\CloudProject> kubectl get pods -n traefik
NAME                       READY   STATUS    RESTARTS   AGE
traefik-5bb6cd4b48-fqmj5   1/1     Running   0          38m
traefik-5bb6cd4b48-pj8ns   1/1     Running   0          38m
PS C:\Users\chris\Desktop\CloudProject> kubectl delete pods/traefik-5bb6cd4b48-pj8ns -n traefik
pod "traefik-5bb6cd4b48-pj8ns" deleted
```



```
PS C:\Users\chris> kubectl get pods -n traefik
NAME                       READY   STATUS    RESTARTS   AGE
traefik-5bb6cd4b48-fqmj5   1/1     Running   0          39m
traefik-5bb6cd4b48-m7zxk   1/1     Running   0          17s
```

A fault can be at pod or container level, but can also be at the application level. Sometimes pods (and the containers within) run without problems even if a fault occurred at the application level (for example in the web server). In order to detect these failures and restore the container (and the application within) health checks are implemented. Kubernetes supports health checks. This means that the **kubelet**, a special agent running on each node (a single one in the case of Minikube), periodically sends requests to the pre-configured health check endpoints of pods on which health checks have been configured. A health check endpoint could be a specific web server path. For example, in this architecture, on each microservice has been defined the ***/health*** endpoint and configured as a health check endpoint. If the web server returns an HTTP status code 200 to the kubelet, the container is considered healthy, otherwise not. If not, the container (not the pod) will be restarted. In the case of this project health checks have been configured in such a way that the kubelet has to receive three error responses (three failed health checks) in order to consider a container unhealthy. Health checks are executed every 10 seconds (in all deployments). For debugging purposes a ***/fail*** endpoint has been defined in order to simulate faults in the web servers and verify that health checks work.

The */health* and */fail* endpoints are the following:

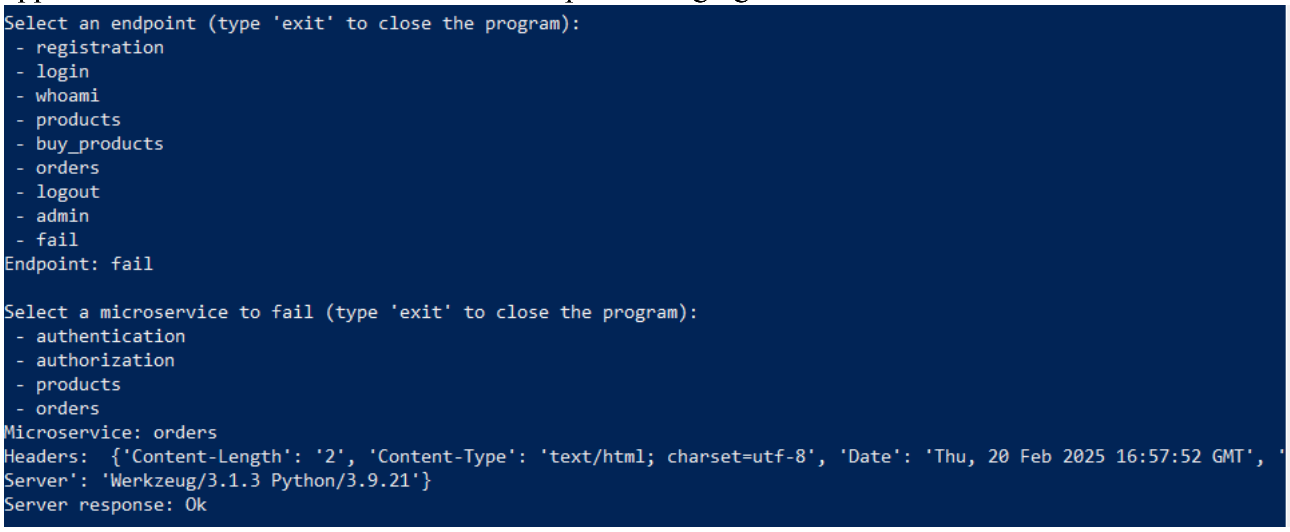


```
@app.route('/health', methods=["GET"])
def isHealthy():
    global healthy
    if healthy:
        return "Healthy", 200
    else:
        return "Unhealthy", 500

@app.route('/authentication/fail', methods=["GET"])
def fail():
    global healthy
    healthy=False
    return "Ok", 200
```

In this image these endpoints are defined in the Authentication microservice, but they are the same on the other microservices as well.

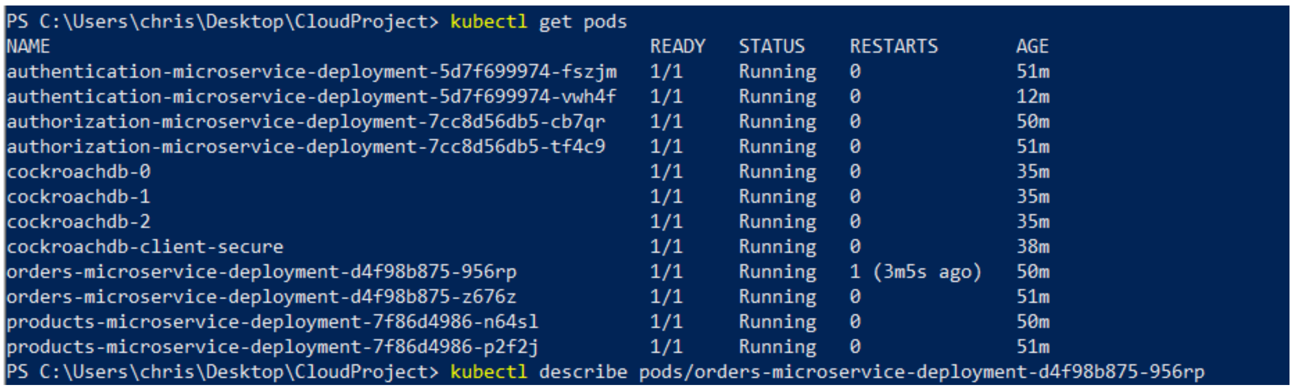
Below an example.

First I executed a request with a client written in Python. This request will cause a simulated application fault in a container situated in a pod belonging to the Orders microservice.



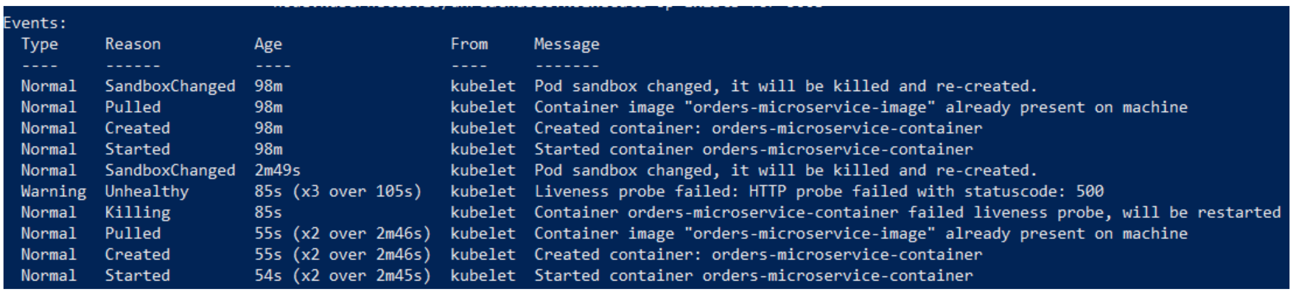
```
Select an endpoint (type 'exit' to close the program):
 - registration
 - login
 - whoami
 - products
 - buy_products
 - orders
 - logout
 - admin
 - fail
Endpoint: fail

Select a microservice to fail (type 'exit' to close the program):
 - authentication
 - authorization
 - products
 - orders
Microservice: orders
Headers:  {'Content-Length': '2', 'Content-Type': 'text/html; charset=utf-8', 'Date': 'Thu, 20 Feb 2025 16:57:52 GMT', '
Server': 'Werkzeug/3.1.3 Python/3.9.21'}
Server response: Ok
```

By checking pods we can see that a pod belonging to the Orders microservices had a container restarted inside it:



```
PS C:\Users\chris\Desktop\CloudProject> kubectl get pods
NAME                                               READY   STATUS    RESTARTS       AGE
authentication-microservice-deployment-5d7f699974-fszjm   1/1     Running   0              51m
authentication-microservice-deployment-5d7f699974-vwh4f   1/1     Running   0              12m
authorization-microservice-deployment-7cc8d56db5-cb7qr    1/1     Running   0              50m
authorization-microservice-deployment-7cc8d56db5-tf4c9    1/1     Running   0              51m
cockroachdb-0                                             1/1     Running   0              35m
cockroachdb-1                                             1/1     Running   0              35m
cockroachdb-2                                             1/1     Running   0              35m
cockroachdb-client-secure                                 1/1     Running   0              38m
orders-microservice-deployment-d4f98b875-956rp            1/1     Running   1 (3m5s ago)   50m
orders-microservice-deployment-d4f98b875-z676z            1/1     Running   0              51m
products-microservice-deployment-7f86d4986-n64sl          1/1     Running   0              50m
products-microservice-deployment-7f86d4986-p2f2j          1/1     Running   0              51m
PS C:\Users\chris\Desktop\CloudProject> kubectl describe pods/orders-microservice-deployment-d4f98b875-956rp
```

If we check the events in that pod we will see the failure of the health checks and the container restart:



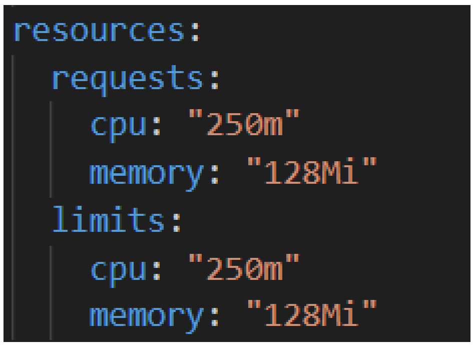
```
Events:
  Type     Reason          Age                  From     Message
  ----     ------          ----                 ----     -------
  Normal   SandboxChanged  98m                  kubelet  Pod sandbox changed, it will be killed and re-created.
  Normal   Pulled          98m                  kubelet  Container image "orders-microservice-image" already present on machine
  Normal   Created         98m                  kubelet  Created container: orders-microservice-container
  Normal   Started         98m                  kubelet  Started container orders-microservice-container
  Normal   SandboxChanged  2m49s                kubelet  Pod sandbox changed, it will be killed and re-created.
  Warning  Unhealthy       85s (x3 over 105s)   kubelet  Liveness probe failed: HTTP probe failed with statuscode: 500
  Normal   Killing         85s                  kubelet  Container orders-microservice-container failed liveness probe, will be restarted
  Normal   Pulled          55s (x2 over 2m46s)  kubelet  Container image "orders-microservice-image" already present on machine
  Normal   Created         55s (x2 over 2m46s)  kubelet  Created container: orders-microservice-container
  Normal   Started         54s (x2 over 2m45s)  kubelet  Started container orders-microservice-container
```

Health checks are also implemented automatically by Traefik (for the API Gateway's pods).

CockroachDB handles automatically fault tolerance. In particular CockroachDB has been configured to have, at least, three CockroachDB nodes (which are pods). CockroachDB documentation advice to use a number of nodes that is a multiple of 3.

Scalability

For each microservice an Horizontal Pod Autoscaler has been configured in such a way that the minimum number of pod replicas is 2 and the maximum is 3 (other values can be configured, I chose these values to save resources). Autoscaling happens whenever the average CPU utilization of the microservice exceeds the 50% (of the associated CPU). This means that the number of replicas for each microservice is initially 2, but as soon as the average CPU utilization of a microservice's replicas exceeds 50% (i.e. on average each pod uses more than 50% of the requested CPU), the HPA associated with that microservice will ad a new pod to that microservice. For testing reasons I set the resource's usage of pods to low values (except for the pods managed by CockroachDB):

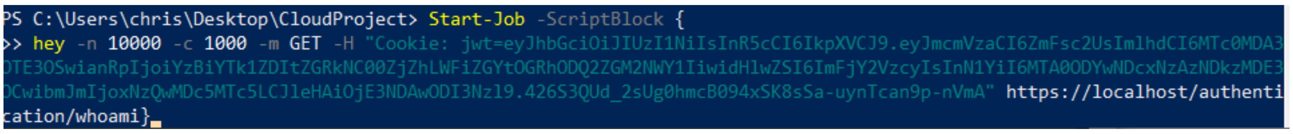


```
resources:
  requests:
    cpu: "250m"
    memory: "128Mi"
  limits:
    cpu: "250m"
    memory: "128Mi"
```

In this way autoscaling can be triggered easily. In many endpoints a *time.sleep(3)* has been added in order to make the request slower for the web server to process.

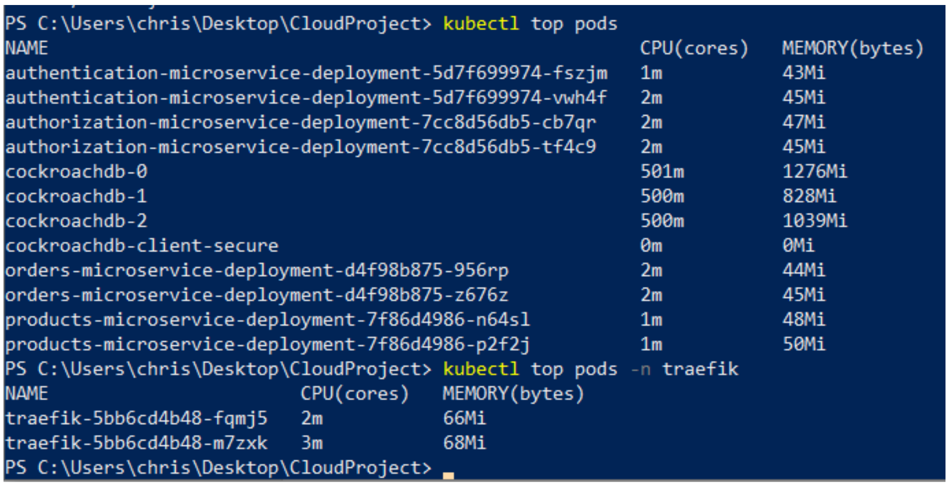
Here is an example of autoscaling when we send a huge number of requests to */whoami*. This request involves the API gateway and the Authentication microservice.

First we execute a command for sending a huge number of requests (10000, divided in groups of 1000 concurrent requests).

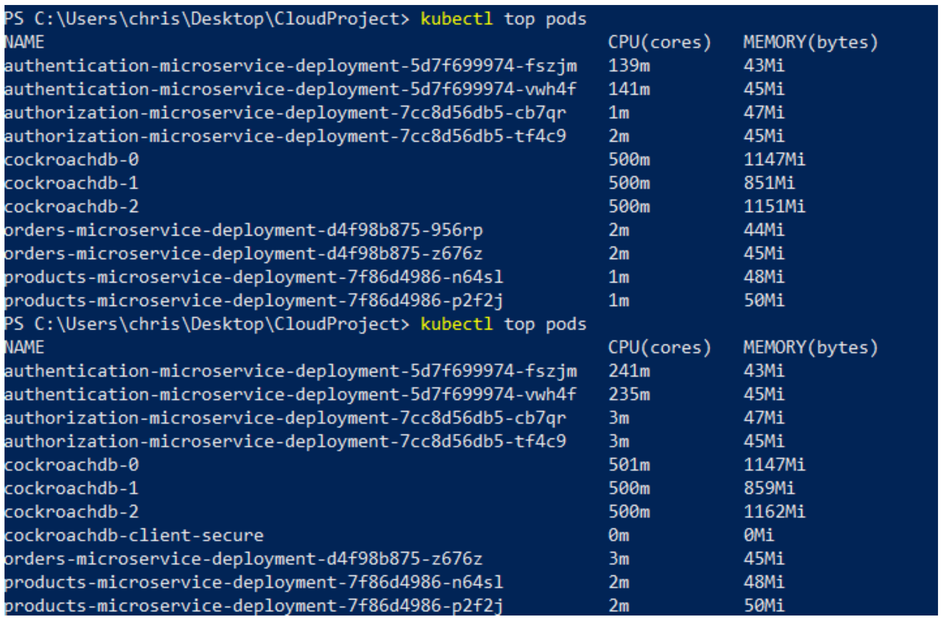


```
PS C:\Users\chris\Desktop\CloudProject> Start-Job -ScriptBlock {
>> hey -n 10000 -c 1000 -m GET -H "Cookie: jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTc0MDA3
DTE3OSwianRpIjoiYzBiYTk1ZDItZGRkNC00ZjZhLWFiZGYtOGRhODQ2ZGM2NWY1IiwidHlwZSI6ImFjY2VzcyIsInN1YiI6MTA0ODYwNDcxNzAzNDkzMDE3
DCwibmJmIjoxNzQwMDc5MTc5LCJleHAiOjE3NDAwODI3Nzl9.426S3QUd_2sUg0hmcB094xSK8sSa-uynTcan9p-nVmA" https://localhost/authenti
cation/whoami}
```

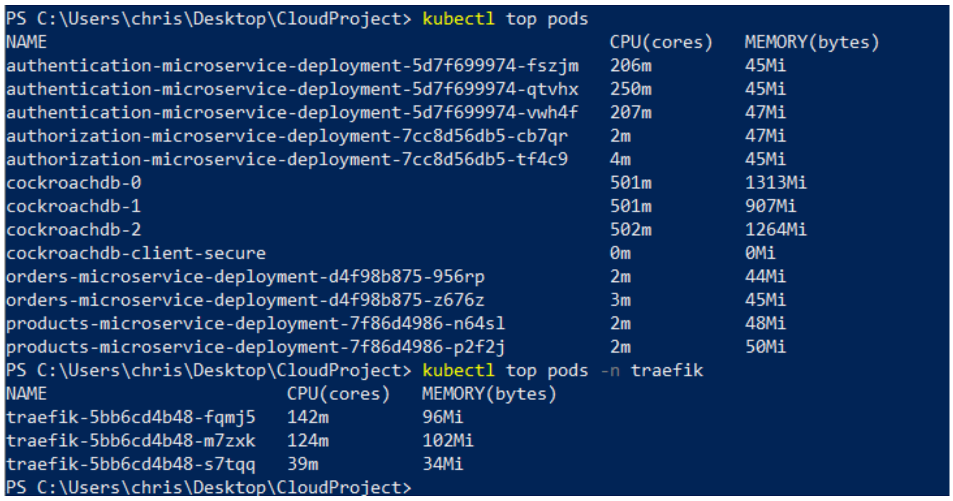
Then we monitor the resources' utilization of pods.



```
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods
NAME                                                 CPU(cores)   MEMORY(bytes)
authentication-microservice-deployment-5d7f699974-fszjm   1m           43Mi
authentication-microservice-deployment-5d7f699974-vwh4f   2m           45Mi
authorization-microservice-deployment-7cc8d56db5-cb7qr    2m           47Mi
authorization-microservice-deployment-7cc8d56db5-tf4c9    2m           45Mi
cockroachdb-0                                             501m         1276Mi
cockroachdb-1                                             500m         828Mi
cockroachdb-2                                             500m         1039Mi
cockroachdb-client-secure                                 0m           0Mi
orders-microservice-deployment-d4f98b875-956rp            2m           44Mi
orders-microservice-deployment-d4f98b875-z676z            2m           45Mi
products-microservice-deployment-7f86d4986-n64sl          1m           48Mi
products-microservice-deployment-7f86d4986-p2f2j          1m           50Mi
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods -n traefik
NAME                        CPU(cores)   MEMORY(bytes)
traefik-5bb6cd4b48-fqmj5    2m           66Mi
traefik-5bb6cd4b48-m7zxk    3m           68Mi
PS C:\Users\chris\Desktop\CloudProject>
```

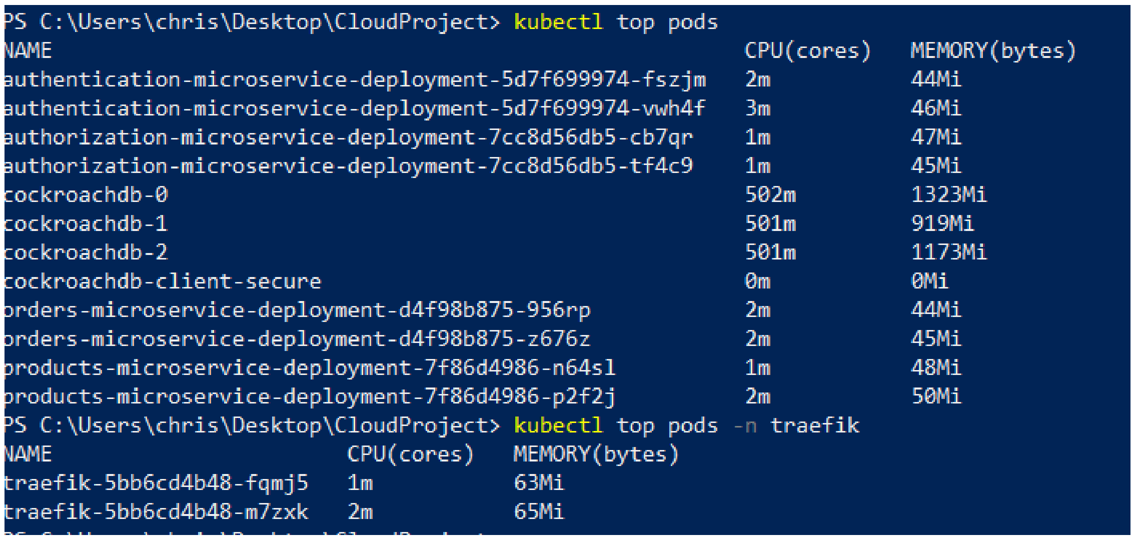


```
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods
NAME                                                CPU(cores)   MEMORY(bytes)
authentication-microservice-deployment-5d7f699974-fszjm   139m         43Mi
authentication-microservice-deployment-5d7f699974-vwh4f   141m         45Mi
authorization-microservice-deployment-7cc8d56db5-cb7qr    1m           47Mi
authorization-microservice-deployment-7cc8d56db5-tf4c9    2m           45Mi
cockroachdb-0                                             500m         1147Mi
cockroachdb-1                                             500m         851Mi
cockroachdb-2                                             500m         1151Mi
orders-microservice-deployment-d4f98b875-956rp            2m           44Mi
orders-microservice-deployment-d4f98b875-z676z            2m           45Mi
products-microservice-deployment-7f86d4986-n64sl          1m           48Mi
products-microservice-deployment-7f86d4986-p2f2j          1m           50Mi
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods
NAME                                                CPU(cores)   MEMORY(bytes)
authentication-microservice-deployment-5d7f699974-fszjm   241m         43Mi
authentication-microservice-deployment-5d7f699974-vwh4f   235m         45Mi
authorization-microservice-deployment-7cc8d56db5-cb7qr    3m           47Mi
authorization-microservice-deployment-7cc8d56db5-tf4c9    3m           45Mi
cockroachdb-0                                             501m         1147Mi
cockroachdb-1                                             500m         859Mi
cockroachdb-2                                             500m         1162Mi
cockroachdb-client-secure                                 0m           0Mi
orders-microservice-deployment-d4f98b875-z676z            3m           45Mi
products-microservice-deployment-7f86d4986-n64sl          2m           48Mi
products-microservice-deployment-7f86d4986-p2f2j          2m           50Mi
```



```
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods
NAME                                                CPU(cores)   MEMORY(bytes)
authentication-microservice-deployment-5d7f699974-fszjm   206m         45Mi
authentication-microservice-deployment-5d7f699974-qtvhx   250m         45Mi
authentication-microservice-deployment-5d7f699974-vwh4f   207m         47Mi
authorization-microservice-deployment-7cc8d56db5-cb7qr    2m           47Mi
authorization-microservice-deployment-7cc8d56db5-tf4c9    4m           45Mi
cockroachdb-0                                             501m         1313Mi
cockroachdb-1                                             501m         907Mi
cockroachdb-2                                             502m         1264Mi
cockroachdb-client-secure                                 0m           0Mi
orders-microservice-deployment-d4f98b875-956rp            2m           44Mi
orders-microservice-deployment-d4f98b875-z676z            3m           45Mi
products-microservice-deployment-7f86d4986-n64sl          2m           48Mi
products-microservice-deployment-7f86d4986-p2f2j          2m           50Mi
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods -n traefik
NAME                      CPU(cores)   MEMORY(bytes)
traefik-5bb6cd4b48-fqmj5   142m         96Mi
traefik-5bb6cd4b48-m7zxk   124m         102Mi
traefik-5bb6cd4b48-s7tqq   39m          34Mi
PS C:\Users\chris\Desktop\CloudProject>
```

As can be seen, overtime pods in the Authentication microservice and in the API gateway increase CPU usage and this leads the HPA of both to add a new pod in both deployments. When CPU utilization will decrease, the useless replicas will be deleted and the number of replicas will be brought back to the minimum number required (in our case 2).

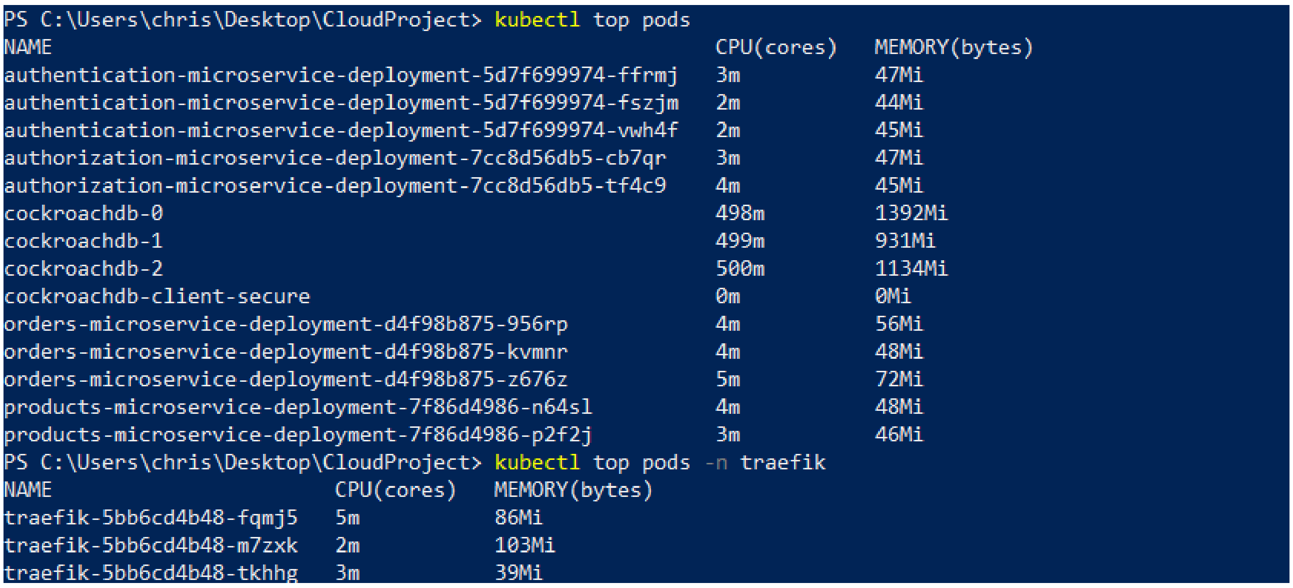


```
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods
NAME                                                CPU(cores)   MEMORY(bytes)
authentication-microservice-deployment-5d7f699974-fszjm   2m           44Mi
authentication-microservice-deployment-5d7f699974-vwh4f   3m           46Mi
authorization-microservice-deployment-7cc8d56db5-cb7qr    1m           47Mi
authorization-microservice-deployment-7cc8d56db5-tf4c9    1m           45Mi
cockroachdb-0                                             502m         1323Mi
cockroachdb-1                                             501m         919Mi
cockroachdb-2                                             501m         1173Mi
cockroachdb-client-secure                                 0m           0Mi
orders-microservice-deployment-d4f98b875-956rp            2m           44Mi
orders-microservice-deployment-d4f98b875-z676z            2m           45Mi
products-microservice-deployment-7f86d4986-n64sl          1m           48Mi
products-microservice-deployment-7f86d4986-p2f2j          2m           50Mi
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods -n traefik
NAME                      CPU(cores)   MEMORY(bytes)
traefik-5bb6cd4b48-fqmj5   1m           63Mi
traefik-5bb6cd4b48-m7zxk   2m           65Mi
```

If we had done the same with */orders*, we would have seen the autoscaling on three deployments:

- API gateway
- Authentication microservice
- Orders microservice

Here's the proof:



```
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods
NAME                                                CPU(cores)   MEMORY(bytes)
authentication-microservice-deployment-5d7f699974-ffrmj   3m           47Mi
authentication-microservice-deployment-5d7f699974-fszjm   2m           44Mi
authentication-microservice-deployment-5d7f699974-vwh4f   2m           45Mi
authorization-microservice-deployment-7cc8d56db5-cb7qr    3m           47Mi
authorization-microservice-deployment-7cc8d56db5-tf4c9    4m           45Mi
cockroachdb-0                                       498m         1392Mi
cockroachdb-1                                       499m         931Mi
cockroachdb-2                                       500m         1134Mi
cockroachdb-client-secure                           0m           0Mi
orders-microservice-deployment-d4f98b875-956rp      4m           56Mi
orders-microservice-deployment-d4f98b875-kvmnr      4m           48Mi
orders-microservice-deployment-d4f98b875-z676z      5m           72Mi
products-microservice-deployment-7f86d4986-n64sl    4m           48Mi
products-microservice-deployment-7f86d4986-p2f2j    3m           46Mi
PS C:\Users\chris\Desktop\CloudProject> kubectl top pods -n traefik
NAME                          CPU(cores)   MEMORY(bytes)
traefik-5bb6cd4b48-fqmj5      5m           86Mi
traefik-5bb6cd4b48-m7zxk      2m           103Mi
traefik-5bb6cd4b48-tkhhg      3m           39Mi
```

After all the traffic has been processed, this one above is the situation.

Load balancing

In this architecture two kind of Kubernetes services are used: **ClusterIP** and **LoadBalancer**. Each microservice is exposed to the others through a ClusterIP service, which automatically implements a **RoundRobin** policy for load balancing between a microservice's pods. The Traefik API Gateway, instead, is exposed through a LoadBalancer Service, which, in Minikube, works the same way as a ClusterIP service. This means that a RoundRobin policy is applied in order to forward requests among the different Traefik pods. CockroachDB also uses a ClusterIP service in order to be exposed to other pods in the cluster.

Traefik API Gateway is also a sort of load balancer, an HTTP(s) load balancer to be exact. For this reason, it's configured to route requests to the correct microservices.

Security

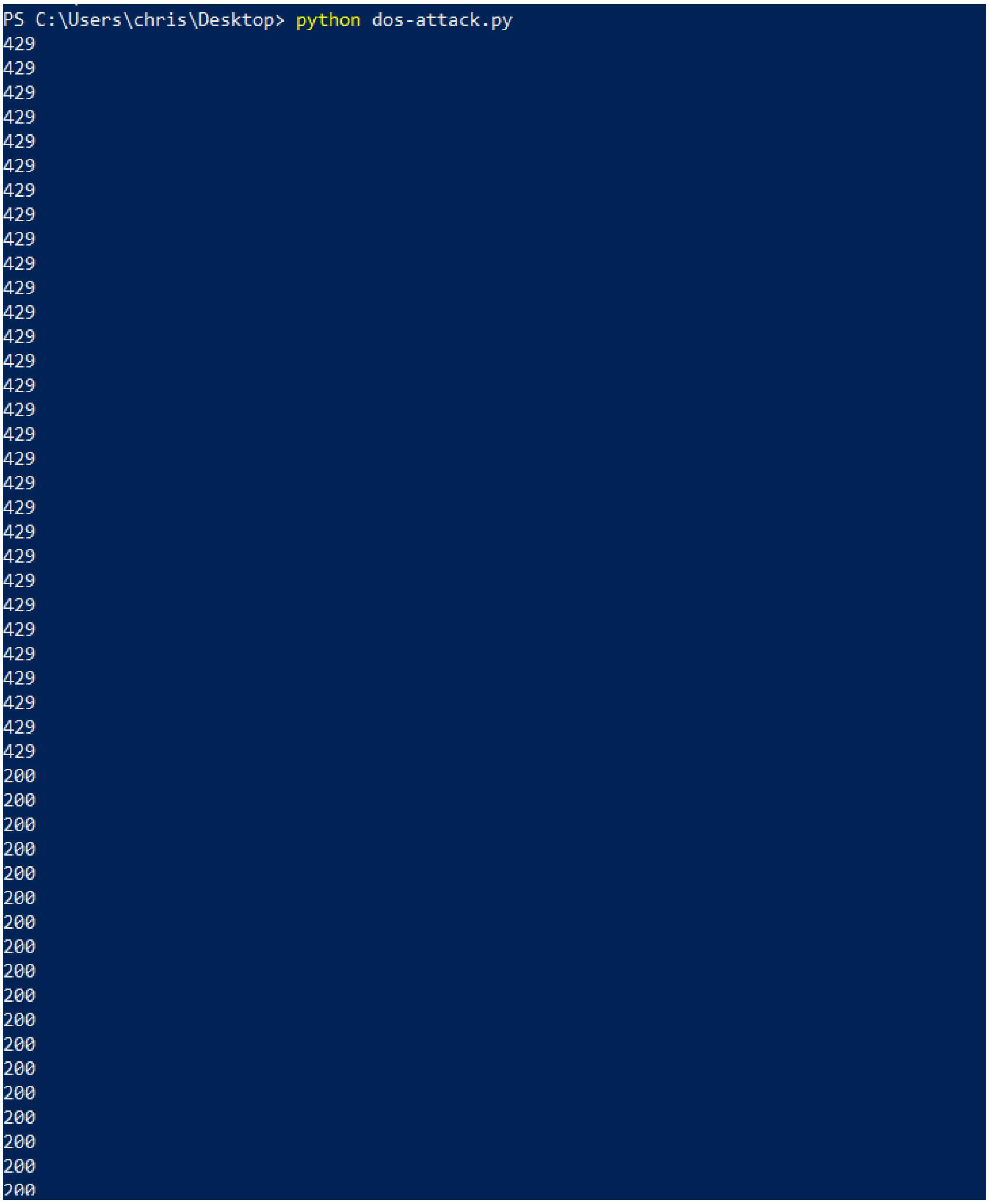
Both API Gateway and microservices use **HTTPS**, thus making all communication encrypted. Obviously, for this project, I used self-signed TLS certificates. Communications with CockroachDB pods (and between them) are automatically encrypted.

DOS attacks are mitigated by *Rate Limit* and *InflightRequests* middlewares configured on the API Gateway on all routes. Traefik's *RateLimit* middleware uses the token bucket algorithm for limiting traffic from the same source. The average number of accepted (i.e. forwarded) requests (incoming from the same source) per minute is set to 20. The burst (i.e. the size of the bucket) is set to 40.

InflightRequests middleware sets the maximum number of simultaneous connections (from the same source) to 40. In case of multiple pods, these values refer to the traffic received by a single pod.

All the requests that exceed these limits (in the pre-configured short period of time, a minute in this case) will not be forwarded to the microservices and will receive HTTP status code 429 in the response.

Here's what happens when a user sends 110 concurrent requests:



```
PS C:\Users\chris\Desktop> python dos-attack.py
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
429
200
200
200
200
200
200
200
200
200
200
200
200
200
200
200
200
200
200
```

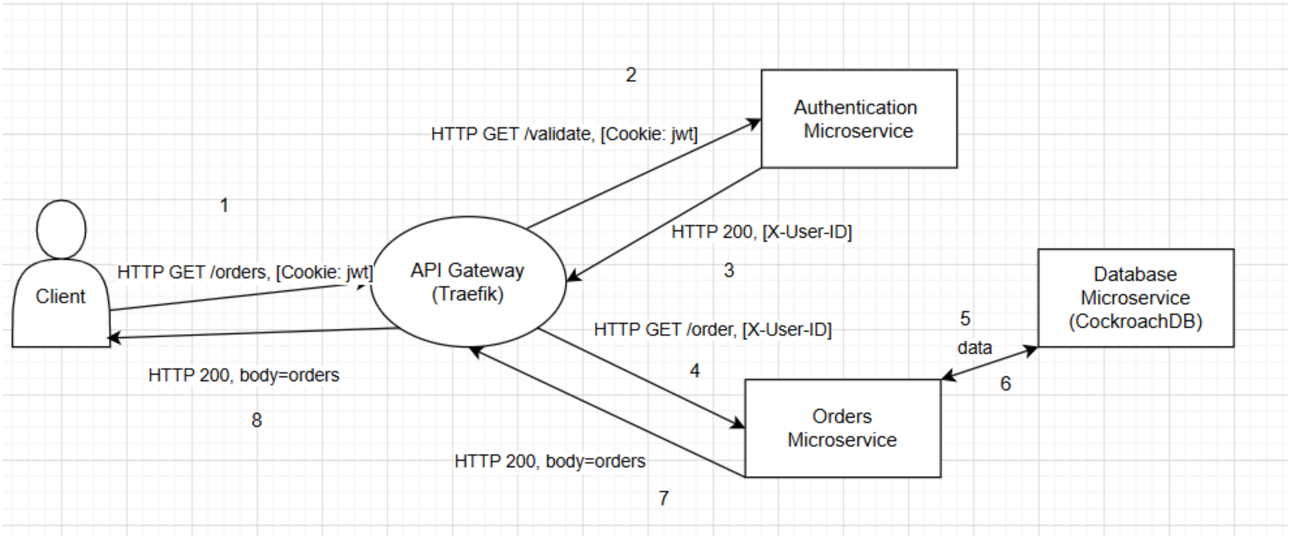
Assuming the token bucket was full, 80 requests were accepted, while the other 30 were denied. This is because there are two pods, so both can handle, when the bucket is full of tokens (at the beginning), 40 requests each (i.e. 80 in total). The remaining requests (30 in this example) will be denied.

With different values, different scenario are possible. This configuration was chosen in order to make testing simpler. These values have been defined for testing purposes only. In real case scenarios other values should be used.

While testing scalability these middlewares have been disabled.

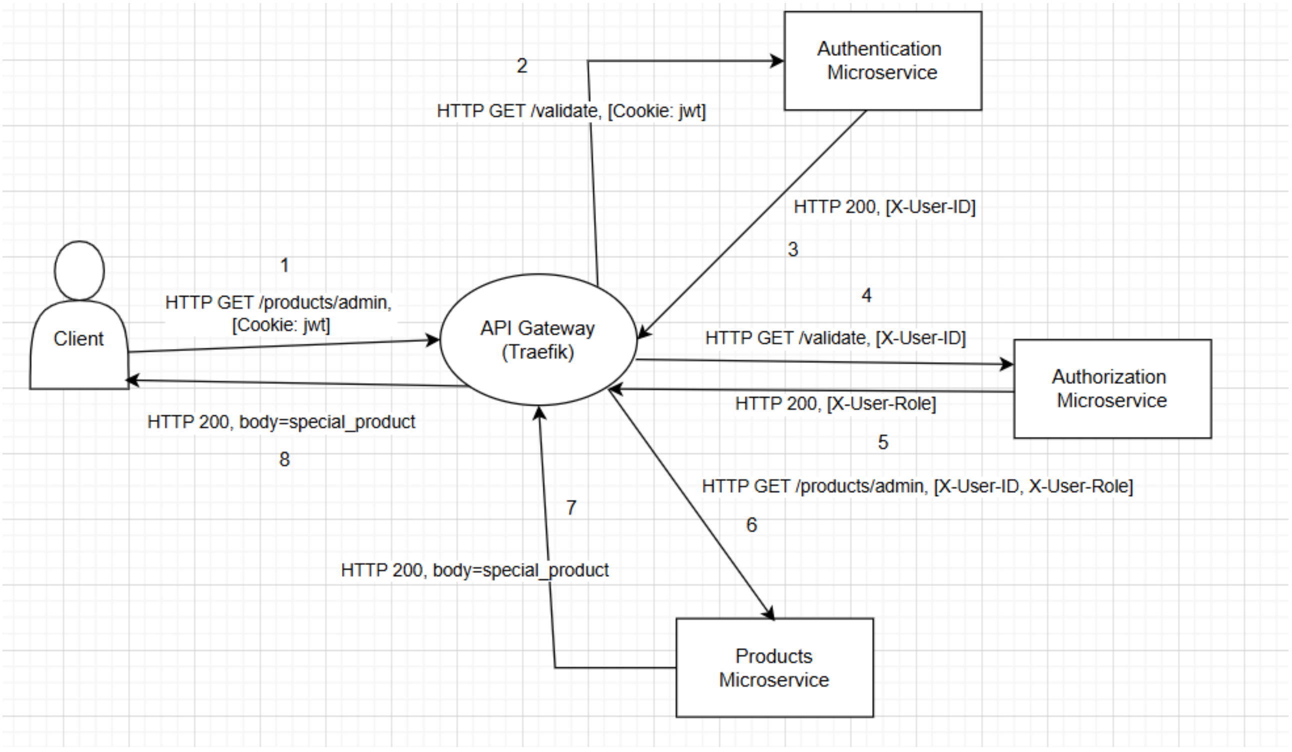
A central aspect of this architecture is the **Authentication** microservice and **Authorization** microservice. Thanks to the *ForwardAuth* middleware the API gateway can be configured to forward specific requests to the Authentication microservice or the Authorization microservice (or both) in order to authenticate and/or authorize requests. Authentication is based on the usage of **JWT**s (JSON Web Tokens).

This is what happens when a request to */orders* is sent to the API gateway:


2
HTTP GET /validate, [Cookie: jwt]
Authentication Microservice
1
HTTP 200, [X-User-ID]
3
Client
HTTP GET /orders, [Cookie: jwt]
API Gateway (Traefik)
Database Microservice (CockroachDB)
HTTP GET /order, [X-User-ID]
5
data
6
4
HTTP 200, body=orders
8
Orders Microservice
HTTP 200, body=orders
7

X-User-ID is a header defined by the Authentication microservice and attached to the user's request in order to enable other microservices to acknowledge user's identity. If the token had been invalid, expired or absent, the Authentication microservice would have returned an HTTP status code 401, which then would have been forwarded to the client.

This is what happens when a request to */products/admin* is sent to the API gateway:


Authentication Microservice
2
HTTP GET /validate, [Cookie: jwt]
HTTP 200, [X-User-ID]
3
1
HTTP GET /products/admin, [Cookie: jwt]
Client
API Gateway (Traefik)
4
HTTP GET /validate, [X-User-ID]
Authorization Microservice
HTTP 200, [X-User-Role]
5
HTTP 200, body=special_product
8
HTTP GET /products/admin, [X-User-ID, X-User-Role]
6
7
HTTP 200, body=special_product
Products Microservice

X-User-Role is a header defined by the Authorization microservice and attached to the user's request in order to enable other microservices to acknowledge user's role. Actually this header is only for information purposes since once a request has been authorized by the Authorization microservice, the other microservices don't check the user's role and simply execute the request (they assume user has the permission to execute that request).

If the request had been authenticated, but the user had not been *admin*, the authorization service would have invalidated the request and the client would have received an error (HTTP status code 401).

Future Works

This architecture is pretty basic. Different configurations are such because the project has been developed in a local environment (Minikube). Different improvements can be applied:

- Deployment on a cloud infrastructure and adaptation of configurations (resources' usage by the pods, autoscaling parameters and so on);
- Usage of mTLS (mutual TLS) instead of TLS in the presentation layer (a possible solution: Istio);
- Usage of a centralized rate limit middleware (a possible solution: Redis);
- Insertion of new microservices.